Broadview
www.broadview.com.cn

[PACKT] open source
PUBLISHING
community experience distilled

Developing Microservices with Node.js

Node.js微服务

本书是以Node.js作为服务端编程语言来讲解微服务开发的实践指南，
通过阅读本书可以帮助你开发出高效且可伸缩的微服务

[美] David Gonzalez 著
赵震一 郑伟杰 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Developing Microservices with Node.js

Node.js微服务

[美] David Gonzalez 著
赵震一 郑伟杰 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书对如何采用 Node.js 及其生态工具进行微服务开发的最佳实践做了全面的介绍，内容包括对微服务架构基本概念及设计原则的讲解，以及如何采用 Node.js 搭配 Seneca、PM2 和 Docker 等现代化工具来构建、测试、监控以及部署轻量级微服务，同时也阐述了 Node.js 在微服务实践中所涉及的相关概念，并就微服务的优缺点、文档化、安全性以及可追溯性等主题进行了探讨。

本书适合掌握服务端开发基本知识的 Node.js 开发者以及使用 Java、C#等其他服务端技术栈并对微服务实践感兴趣的所有开发者。

版权贸易合同登记号　图字：01-2016-7164

图书在版编目（CIP）数据

Node.js 微服务/（美）大卫·冈萨雷斯（David Gonzalez）著；赵震一，郑伟杰译. —北京：电子工业出版社，2017.1

书名原文：Developing Microservices with Node.js

ISBN 978-7-121-30524-5

Ⅰ. ①N… Ⅱ. ①大… ②赵… ③郑… Ⅲ. ①JAVA 语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2016）第 290015 号

策划编辑：张春雨
责任编辑：刘　舫
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：16　　字数：320 千字
版　　次：2017 年 1 月第 1 版
印　　次：2017 年 1 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

译者序

在技术圈里，微服务已不是一张生面孔，确切地说，如今已算得上是一名当红的明星了。然而，对于任何一门流行的技术而言，从出现到被广泛采纳必定都会经历一个被质疑、被挑战，以及在实践的锤炼中不断进化的过程，微服务也是如此。

作为本书的译者，我并不打算在这篇序里给出太多的剧透。但是对扒一扒“微服务”的成长史却深感义不容辞。

根据 Martin Fowler 大叔的回忆，“Microservices”一词是在 2011 年 5 月于威尼斯附近举办的一次架构师工作坊的讨论中被首次提出的。

2012 年，微服务正式出道。“Microservices”一词首次在 ThoughtWorks 技术雷达 2012 年 3 月的报告中亮相。当时报告对其成熟度的评级位于“评估（Assess）”象限。

不到一年时间，“Microservices”一词在 2012 年 10 月的技术雷达中已经进入了“试验（Trial）”象限。这份报告称，TW 及更广范围内的社区都将微服务作为一项分布式系统设计的技术开始采用。

2013 年，可配合微服务实施的一些框架和工具相继出现，比如 Spring Boot、Hystrix 等。

在此之后，业界对于微服务的实践及讨论逐渐升温。2014 年 3 月，Adrian Cockcroft（前 Netflix 首席云架构师，被誉为“让 Netflix 走向云端的男人”）与 John Allspaw（现任 Etsy 的 CTO）等人在 Twitter 上展开了关于“微服务与单块应用”孰优孰劣的讨论[1]。

而在业界，真正为“微服务架构”这一架构风格正名的当属 Martin Fowler 大叔于 2014

[1] https://www.infoq.com/news/2014/08/microservices-monoliths

年 3 月在其博客发表的 *Microservices*[2]一文，也正是此文让大众对微服务有了更加具体的认识。

2015 年，随着以 Docker 为代表的容器技术的突飞猛进，微服务的部署难题迎刃而解，甚至有人将 2015 年称为微服务架构元年。

而当我们跨入 2016 年甚至是 2017 年的时候，微服务已正值壮年。在书店及互联网上，关于什么是微服务的书籍、博文已成燎原之势。对于那些希望了解微服务“是什么”的人来说，这是一个美好的时代。但是就微服务生命周期各个阶段该“怎么做”而言，译者深感始终缺少一本接地气的实践指南。

当电子工业出版社计算机出版分社的张春雨编辑向我推荐这本书的时候，我的心里是纠结的。因为我目前本职的工作量是相当饱和的（老板请看过来~），但是却无法拒绝这本书。这不就是那部我寻找已久的接地气的“missing guide”吗？

尤其是将“微服务”与“Node.js”这两味如此珍贵的药材一起入药时，它们又会对“单块系统”中的哪些痼疾产生怎样奇特的疗效呢？说好了不剧透，那么就请读者亲自体会这一段接地气的技术之旅吧。

再次感谢张春雨编辑对我的信任，也非常荣幸能参与这本书的翻译。当然，翻译一本书并不是一件轻松的事情，我要感谢我的师弟郑伟杰，他与我共同承担了本书的翻译工作，正因为有了他的加入才让我得以工作翻译两不误。其次要感谢伟杰的女友，她为本书译文做了审阅与润色。最后，我要感谢我的家人，尤其是我的老婆和父母，你们是我坚强的后盾，让我能专注于做好自己喜欢的事情。

由于时间及能力所限，我们对于原书的理解及对译文的表述难免存在一些不妥之处，希望各位读者给予理解及反馈。我的邮箱是 emsn1026@gmail.com，欢迎各位读者与我们联系。

赵震一

2016 年 8 月于杭州

[2] http://martinfowler.com/articles/microservices.html

关于作者

David Gonzalez 是一名在编程语言方面“极不专一”的软件工程师，他在金融服务行业“混迹”多年。他尝试找到抽象层次合适的解决方案，并探索着如何保证既不过于具体也不过于抽象之间的平衡。

David 曾求学于西班牙，但是不久之后便转战都柏林，自 2011 年起便定居于此并开启了更为宽广和有趣的职业生涯。他目前是一名金融技术领域的独立咨询师。他的 Linkedin 账号地址是：`https://ie.linkedin.com/in/david-gonzalez-737b7383`。

David 乐于尝试新的技术和范式，从而能让自己在软件开发的复杂世界中不断拓展出新的版图。

献给我的妻子 Ester，感谢你在生活中的方方面面都给予我无条件的支持。

献给我尚未出生的女儿 Elena，愿生活带给你所有的快乐，一如你会将这些快乐同样带给我们全家。

关于审校者

Kishore Kumar Yekkanti 是一名经验丰富的专家，他在过去的十年里曾与不同的领域和技术打过交道。他对软件开发中的消除浪费尤具热情。Kishore 是敏捷原则的巨大贡献者和遵循者。他是一名善于开发端到端系统的全栈开发者，同时也是一名通晓多种语言的程序员。目前他专注于高度分布式应用中的微服务扩展，而这些应用部署于云端基于容器的系统(Docker)之中。他曾在多家知名的公司担任过首席工程师，这些公司包括 Thoughtworks、CurrencyFair 等。他曾通过微服务为这些公司的团队带来新生。

献给我的搭档和挚友 Jyothsna，以及我的女儿 Dhruti，尽管我的工作日程安排得是那么疯狂，她却一如既往地迁就着我。

前言

作为一本微服务入门的实践指南，本书采用了 Node.js 和以 Seneca、PM2 为主的现代框架来进行阐述。在各章中，我们将分别介绍如何利用最佳实践去设计、构建、测试以及部署微服务。此外，我们还会讨论另外一个有价值的课题——如何在设计系统时做出合理的妥协，来避免过度设计和确保技术方案与实际业务需求相匹配。

本书概述

第 1 章主要讲述微服务的基本概念，包括主要优点和一些缺点，本章内容是本书后续章节的基础。

第 2 章介绍了 Node.js、Seneca 和 PM2。还讨论了 Node.js 应用的结构，以及如何通过 PM2 来运行应用。另外，我们还研究了一些 Seneca 与 PM2 的替代产品。

第 3 章主要讲述如何使用微服务来处理自然增长（计划之外的软件需求变更）。另外，我们还讨论了如何将单块应用分解成微服务。

第 4 章阐述了如何编写我们的第一个微服务程序。

第 5 章涉及了安全性与可追溯性，这是现代系统的两大重要特性，因为我们需要保证信息的安全与操作的可追溯性。在本章中，我们讨论了使用 Seneca 来保证安全性与可追溯性的方法。

第 6 章主要介绍了 Node.js 的两大主流测试框架——Mocha 和 Chai。同时使用 Sinon 来 mock 服务以及 Swagger 来为微服务进行文档化。

第 7 章使用 PM2 结合 Keymetrics 来监控微服务，使 PM2 的功能得到最大发挥。

第 8 章通过使用 PM2，学习如何在不同环境下部署微服务，并通过单条命令管理应用的“生态系统”，从而减少微服务架构带来的开销。我们还将讨论 Docker，它是一个应用容器引擎，可以部署包括 Node 应用在内的各种应用。

阅读本书的准备工作

为了能够完成本书的实践案例，需要预先安装 Node.js、PM2（可以通过 npm 来安装），以及 MongoDB。此外还需要一个编辑器，我个人选用了 Atom，但是一般通用的编辑器都能满足需求。

本书的读者对象

本书适合具有一定 Node.js 经验，并且想要学习 Seneca 以及微服务知识的开发者。在本书中，有 70%的内容是面向实践的（因此我们会编写大量代码），有 30%是理论知识。基于编写的这些代码可以帮助读者将书中提到的模式应用到新的软件开发中去。

约定惯例

本书将会使用不同的书写风格来区分不同种类的信息。以下是这些风格的例子和它们的意义。

正文中的文本代码、数据库表名、文件夹名、文件名、文件扩展名、路径名、URL、用户输入和推特用户定位（Twitter handles）将会用代码体书写，如“我们知道输入参数是一个 `PaymentRequest` 实例”。

代码块则将会是这样的风格：

```
public interface PaymentService {
   PaymentResponse processPayment(PaymentRequest request) throws
      MyBusinessException;
}
```

如果希望向你强调代码块中的一部分，那么它们将会以粗体展示：

```
function() {
```

```
    sinon.stub(Math, "random");
    rollDice();
    console.log(Math.random.calledWith());
});
after(function(){
    Math.random.restore();
});
```

任何命令行的输入和输出将是以下这样的：

```
node index.js
npm start
```

警告和关键提醒将会在这样的图标后出现。

小提示和小技巧将会在这样的图标后出现。

下载示例代码

你可以从 http://www.broadview.com.cn 下载所有已购买的博文视点书籍的示例代码文件。

勘误表

虽然我们已经尽力谨慎地确保内容的准确性，但错误仍然存在。如果你发现了书中的错误，包括正文和代码中的错误，请告诉我们，我们会非常感激你。这样，你不仅帮助了其他读者，也帮助我们改进后续的出版。如发现任何勘误，可以在博文视点网站相应图书的页面提交勘误信息。一旦你找到的错误被证实，你提交的信息就会被接受，我们的网站也会发布这些勘误信息。你可以随时浏览图书页面，查看已发布的勘误信息。

目录

1 微服务架构

微服务日趋流行。时下，几乎每一个参与“绿地项目”[译注1]的工程师都应该考虑采用微服务来提升所建系统的质量，他们应该熟悉涉及这类系统的架构原则。我们将会在本书中揭示微服务与面向服务架构（SOA）之间的区别。同时还会向大家介绍一款用以编写微服务的强大平台——Node.js。通过采用 Node.js，我们可以轻易地创建出高性能的微服务。

本章将会带领你从架构的视角来审视微服务：

- 什么是微服务
- 面向微服务的架构
- 关键的好处
- SOA 与微服务的比较
- 为什么选择 Node.js

微服务应运而生

在过去 40 年里，软件开发的世界日新月异。在这一飞速发展的过程中，一个关键的增长点便是系统的规模。从古老的 MS-DOS 发展到当代的系统，规模有了数百倍的增长。这种飞跃式的增长迫使我们不得不去寻求更好的方式来组织代码及软件组件。通常，在一家公司随着业务需求的增长而逐步发展（即自然增长）的过程中，前期往往是以单块架构（monolithic architecture）的方式来组织系统的。因为对于软件的初期构建来说，单块架构

译注1 绿地项目（green field）意指那些没有太多前置约束的项目。类比自在一片绿地上开发项目，没有既有建筑与基础设施的约束。

的方式是最容易且最高效的。但是若干年（甚至是几个月）后，受限于前期既有单块软件系统内部的耦合性，向该系统添加新功能变得越来越艰难。

单块软件

如今，对于像 Amazon 和 Netflix 这样的高新技术公司来说，采用微服务来构建新的软件系统已经是大势所趋。理想情况下，面向微服务的软件在帮助这些公司扩展新产品规模方面有着强大的优势（学习完本书后，你也将深谙此道）。现在的问题是，并不是所有的公司都能提前对软件系统做好规划。相比于提前规划，很多公司一直以来都是基于自然增长的方式来构建软件系统的，鲜有软件组件会根据业务关系的紧密度来对业务流进行分类。不难发现，大多数公司都只有两个大的软件组件：面向用户的网站和内部的管理系统。我们通常将这种架构方式称为*单块软件架构*。

一部分这类公司在尝试扩充工程团队的时候，面临了巨大的挑战。协调好多个团队来对单个软件组件进行构建、部署以及维护是一件相当艰巨的任务。系统发布与 bug 重复引入之间的冲突往往是家常便饭，这些问题耗费了团队大量的精力。解决该问题的一个方案（同时还会带来额外的好处）是将单块软件拆分成微服务。每个团队都专职从事某个相对较小模块的维护，且这些组件也都是自治且互相隔离的。这样一来，我们便可以独立地对这些组件进行版本化、更新以及部署，从而不会牵扯公司的其他系统。

将庞大的单块系统拆分成微服务可以让工程团队创造出互相隔离且独立自治的工作单元。这些工作单元从功能上来说都是高度专职化的，例如可以专职处理电子邮件的发送、支付卡交易等特定任务。

现实世界中的微服务

微服务是指那些小型的软件组件，每个服务都专职处理某一类任务，将它们有机整合后便可用于处理更高级的复杂任务。让我们先忘掉软件这个概念，然后回过头来思考一下一家公司是如何运作的。当某个候选人申请公司的某份工作时，他应聘的是一个特定的职位，比如：软件工程师、系统管理员或者办公室经理。之所以这样的原因归根结底就是一个词：专职化（specialization）。如果你曾任职软件工程师，将会在软件开发方面具有更丰富的经验，并且能在这方面给公司带来更多的价值。事实上，你并不知道应该如何跟客户打好交道，但是这并不会影响你的绩效，因为这并不是你的专业领域，所以在这方面你很难在日常工作中产出更多的价值。

专职化通常是提升效率的关键。做一件事并将它做对是软件开发的准则之一。

微服务是一个自治的工作单元，它可以执行某个任务且不干涉系统的其他部分，就好比公司中的某个专职的工作职位。这种方式可以带来很多好处，它有助于工程团队对公司的系统规模进行扩展。

如今，数以百计的系统都采用了面向微服务的架构来构建，我们来看看：

- Netflix 是时下最流行的流媒体服务商之一，它构建了囊括自身所有应用的整个生态系统，这些应用互相协作，从而构建出能为全球范围提供可靠和可伸缩服务的流媒体系统。
- Spotify 是世界领先的音乐流媒体服务商，它采用了微服务来构建应用。它的应用（该应用是一个采用 Chromium Embedded Framework 实现的以桌面方式呈现的网站）中的每个单独的小部件都是一个可以独立升级的微服务。

面向微服务的架构

面向微服务的架构拥有一些特质。任意规模的大中型公司如果想让自身 IT 系统保持弹性，并希望随时都可以对系统规模进行伸缩的话，必定会对这些特质趋之若鹜。

为什么面向微服务的架构更好

面向微服务的架构并不是软件工程的圣杯。但是，对于那些依赖于技术而发展的公司而言，如果能将该架构运用恰当，那么将会是解决这些公司所面临的大部分重要问题的完美方法。

弹性、可组合性以及灵活性都是面向微服务架构设计的关键原则。将这些原则烂熟于胸是一件非常重要的事情，否则，你将错失一个完美的解决方案，并最终在将单块应用分拆到多台机器上时遇到大量的问题。

不足之处

事无绝对，周遭也存在着一些针对面向微服务架构的诟病。大家都提到了一些需要处理的问题，例如：延迟、可追踪性以及在单块软件中并不存在的配置管理的问题。其中一

些问题描述如下。

- **网络延迟**：微服务具有分布式的特性，从而无可避免地会存在网络延迟。
- **运维负担**：更多的服务器也意味着需要更多的维护工作。
- **最终一致性**：在一个对事务性要求较高的系统中，考虑到实现的局限性，各个节点在某一段时间内可能会出现数据不一致（我们将会在本章的后续部分继续讨论这一点）。

总而言之，工程师应该尽可能对该方法的利弊做出评估，然后根据是否满足相应业务需求来做出是否采用微服务的决定。

面向微服务的架构存在着一些需要被考量的特质。当软件工程师在编写单块软件的时候，受限于所构建软件的特性，很多问题都被忽视了。

举个例子，想象一下某个软件正需要一个发送电子邮件的功能。在单块软件中，我们会在应用的核心部分加入该功能的代码。更有甚者会选择创建一个专门用于处理电子邮件的模块（听上去貌似是个好主意）。而现在，我们并不打算向这个庞大的软件组件添加功能，而是选择了创建一个微服务。这是一个可以独立部署且可以独立版本化的专用服务。在这个案例中，我们忽略了一件事，即：在请求该新的微服务的过程中会产生网络延迟。

在前面的例子中，不管你采用哪种方式（单块方式亦或是微服务）来构建软件，并不存在任何严重的问题；举个例子，就算丢失了一封邮件，也不会因此就到达世界末日。根据定义，电子邮件的传递是不可靠的。所以即便收到一些来自客户的投诉，你的应用也将会继续运作下去。

关键设计原则

下面是一些在构建微服务时需要考虑的关键设计原则。由于微服务是一个新概念，所以目前并没有任何现成的“黄金法则”。因而在实践的过程中，往往会缺少一些可以遵照的共识。

一般来说，我们可以确定以下设计原则：

- 微服务是一系列参考公司流程模型而分拆出来的业务单元。
- 它们是一系列包含了业务逻辑并能采用简单信道和协议与之进行通信的智能端点。
- 根据定义，面向微服务的架构是去中心化的，从而可以构建出健壮和具有弹性的软件。

从组件到业务单元

在软件工程中，创建新项目是一件大家喜闻乐见的事情。因为在新项目中，你可以肆意发挥全部的创造性，并在其中尝试各种新的架构理念、框架以及方法论。然而不幸的是，对于一家成熟的公司来说这并不是一种常态。通常你需要做的就是在一个既有的软件系统中添加新的组件。在创建新组件的时候，有一个最佳的设计原则值得大家遵守：就是要尽可能保持组件与软件其他部分的低耦合，这样它才能作为一个独立的单元进行工作。

保持较低的耦合度得以让我们毫不费力地将软件组件转换成微服务。

让我们来看一个实际的例子：公司的应用目前急需处理支付的功能。

通常我们会为此创建一个新的模块，该模块可以处理与所选支付厂商（例如信用卡、PayPal 等）之间的相关事务，然后会将所有与支付相关的业务逻辑封装在该模块中。定义的接口代码如下：

```
public interface PaymentService {
  PaymentResponse processPayment(PaymentRequest request) throws
    MyBusinessException;
}
```

这是一个大家都能理解的简单接口，而它却是通往微服务世界的关键。我们已经将所有的业务知识封装在该接口背后，所以理论上可以任意切换支付厂商而不会影响应用的其他部分。对外面的世界来说，接口内部的实现细节是不可见的。

到目前为止我们掌握了如下信息：

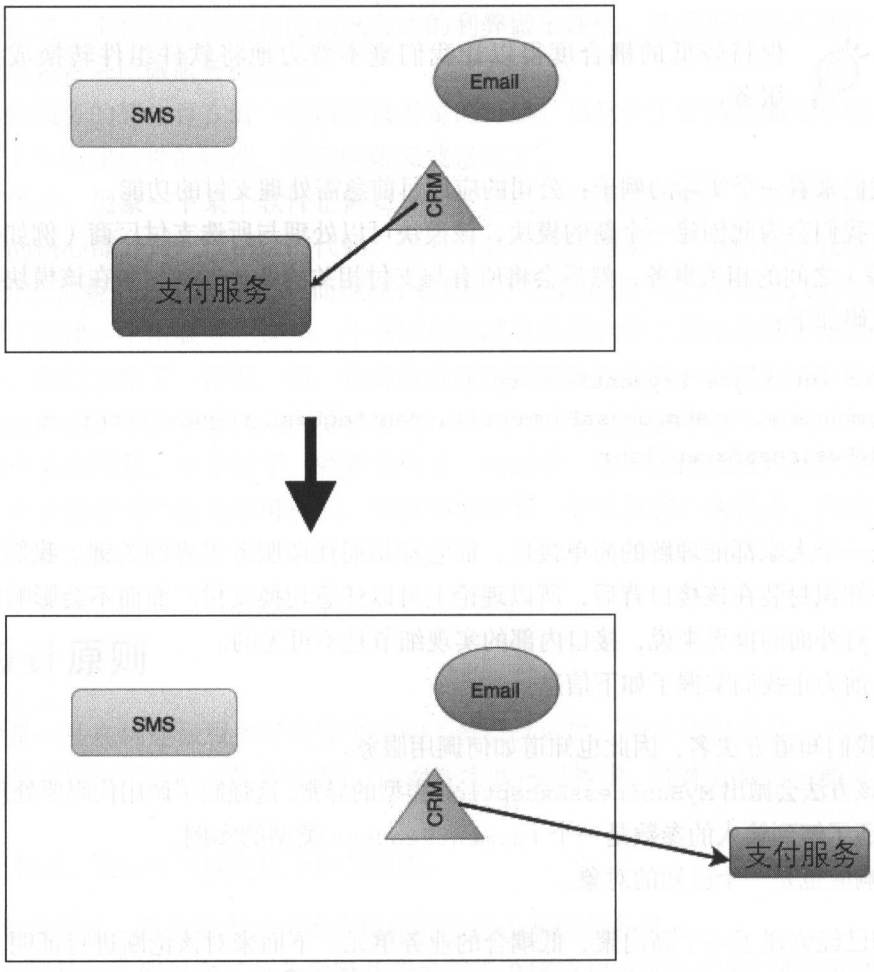
- 我们知道方法名，因此也知道如何调用服务。
- 该方法会抛出 `MyBusinessException` 类型的异常，这强制了调用代码要处理该异常。
- 还了解到输入的参数是一个 `PaymentRequest` 类型的实例。
- 响应也是一个已知的对象。

我们已经创建了一个高内聚、低耦合的业务单元。下面来对该论断进行证明。

- **高内聚：**所有支付模块内的代码只做了一件事，就是处理支付以及调用第三方服务（比如信用卡处理器）相关的事项（连接处理、响应代码等）。

- **低耦合：**如果出于某些原因，需要切换到一个新的支付处理器会怎么样？接口本身是否会泄露出任何实现细节的信息？如果接口的契约发生了变化，是否需要更改调用服务的代码？这些问题的答案当然是否定的。支付服务接口的实现将永远是一个模块化的工作单元。

下图展示了一个系统是如何将其内部的一个组件（支付服务）剥离成微服务的，该系统由多个组件组成：



一旦实现该模块，我们便拥有了处理支付的能力。而该单块应用的这项功能也已经具备了被提炼成微服务的能力。

现在，我们将揭开新版支付服务的面纱。只要接口不发生变化，与外界（系统或第三方的服务）交互的契约也将保持不变。这也是为什么保持实现与接口之间独立性如此重要的原因，即使你所使用的语言并不支持接口。

为了满足业务需求，我们同样也可以根据自身需要来部署更多的支付服务从而扩大其规模。而对于应用中的其余模块而言，由于暂无太大的访问压力，可暂不扩容。

智能的服务，愚蠢的通信管道

超文本转移协议（HTTP）是伴随互联网而生的最棒的工具之一。人们将 HTTP 设计为无状态，但又通过巧妙地结合 cookies，从而让客户端可以保持状态。这些事出现在 Web 1.0 时代，那时还没有人谈论 REST API 和移动应用。让我们来看一个 HTTP 请求的例子：

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
ETag: "3f80f-1b6-3e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 138
Accept-Ranges: bytes
Connection: close
```

正如你所看到的，这是一个人类可读的协议，无须太多解释就能被人理解。

如今，大家都已经明白，HTTP 并不局限于在 Web 中使用。正如它的作者们所设计的，它目前已经被当作通用协议来使用，可用于在两个端点之间传输数据。HTTP 正是我们所需要的、可用于微服务之间进行通信的协议：它可以传输数据，并在传输出错的情况下尽可能完成自我修复。

在过去的若干年里，尤其是在企业内部，人们花费了大量的精力来创建智能的通信机制，例如 BPEL。BPEL 是*业务流程执行语言*（Business Process Execution Language）的简称，它专注围绕各业务环节之间的衔接行为，而非通信行为本身。

这样一来便在通信协议中引入了一定程度的复杂度，使得应用的业务逻辑从端点渗透到了通信协议之中，从而导致了端点之间出现了一定程度的耦合。

应该将业务逻辑限定在各个端点之内，而不是任由其渗透到通信信道之中，这样的系统更便于测试与扩展。多年来，我们认识到，通信层必须是一个能确保数据在端点（微服务）之间有效传输且相对扁平、简单的协议。我们应该将业务逻辑封装在端点的实现中，因为在实际情况中，一个服务并不能保证每时每刻都在线（应对这一状况的能力被称为弹性，会在本章的后续部分继续讨论），而网络也可能会时不时出现一些通信问题。

HTTP 通常是用以构建面向微服务的软件的最广泛的协议，而另一个值得我们关注且需要进一步探索的选项是使用队列来简化端点之间的通信，比如 Rabbit MQ 和 Kafka。

队列技术以缓冲的形式为我们提供了一种管理通信的全新方式，它对高度事务性系统中消息确认机制的复杂性进行了封装。

去中心化

单块应用的一个主要缺点就是将所有的事物集中在单个（或少量）软件组件和数据库中。这样做多半会产生超大的数据存储以及对工作流的集中式控制。而该数据存储是需要根据公司业务成长的需求不断复制与扩容的。

微服务的目标是去中心化。相比于构建超大的数据库，它选择根据前文提到的业务单元来对数据进行拆分。

一些读者会将事务性作为主要的理由从而拒绝使用微服务。先来看看下面的场景：

1. 一个客户在面向微服务的网店中购买了一件商品。
2. 当进入支付环节时，系统发出了如下调用：
   （1）向公司的财务系统发起请求，创建了一个支付的事务。
   （2）向仓储系统发起请求，通知对客户购买的书籍进行发货操作。
   （3）向邮件系统发起请求，为客户订阅新闻邮件。

在一个单块软件中，所有的调用都被包装在一个事务中。因此，如果有某些原因导致任何一个调用失败，其他调用涉及的所有数据都不会持久化到数据库中。

当你开始学习数据库设计时，首先接触到的一个重要原则便是 ACID，它是以下四个属性的缩写。

- **原子性：**每一个事务要么全部生效，要么全部回滚。只要有一部分失败，不会有任何变更持久化到数据库中。
- **一致性：**事务中产生的数据变更会得到一致性保证。
- **隔离性：**事务并发执行产生的系统状态将与事务串行执行产生的状态保持一致。

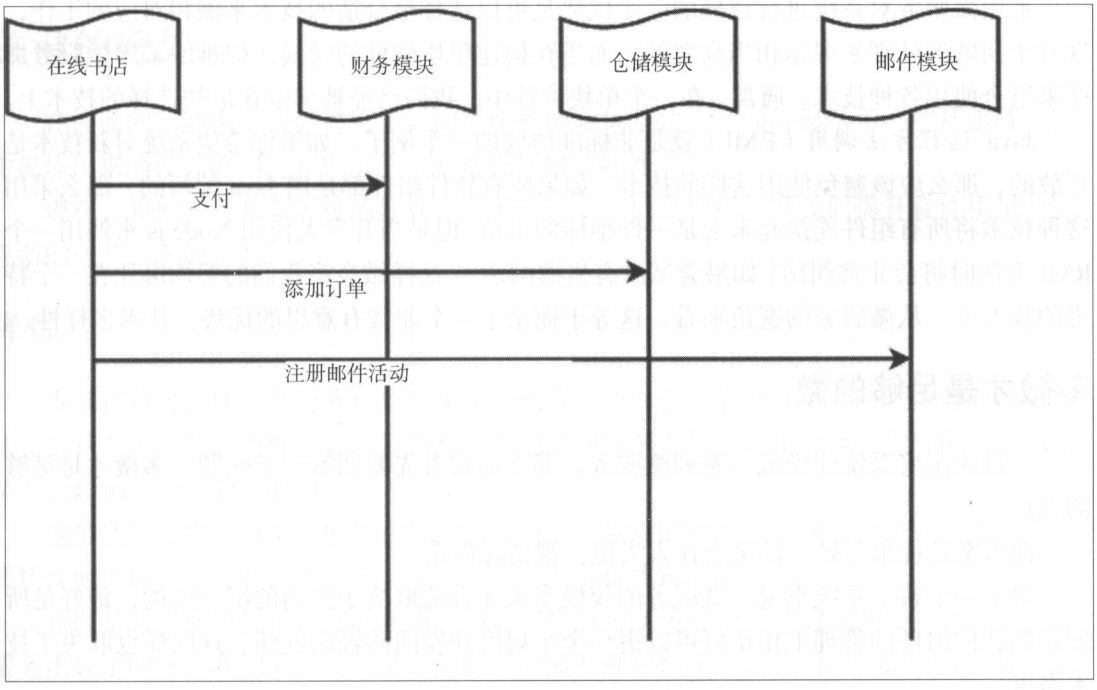
- **持久性**：一旦事务被提交，数据将被持久化。

在一个面向微服务的软件中，ACID 原则是无法得到全局保证的。微服务将会在本地提交事务，但是并没有机制能保证100%完整的全局事务。在上面的场景中，可能会出现没有处理支付就已经发货的情况，除非我们提前制定了特定的规则来防止这样的情况发生。

在一个面向微服务的架构中，数据的事务性是无法保证的，所以我们要在实现中考虑到失败的因素。一种解决该问题的方式（或者称之为一种变通的方案更为合适）是将管理与数据存储去中心化。

在构建微服务的时候，我们需要对下面的因素进行考量并将应对方案纳入设计之中。即一个或多个组件可能会发生失败，而我们需要根据软件的可用性来对功能进行降级。

让我们来看看下面这幅图：



该图表示了单块软件中的一组执行序列。不管怎样，这一系列调用的执行都应当遵守ACID 原则：要么所有调用（事务）都成功，要么什么都不做。

为此，框架和数据库引擎的设计者提出了事务的概念来保障数据的事务性。

当我们与微服务打交道时，不得不提及一个概念，工程师们称之为最终一致性。当事务的一部分发生失败后，每个微服务实例都应该将用于恢复事务的信息存储下来，以便于

这些信息能达到最终一致性。根据前面提到的例子，如果我们对书籍执行发货操作而没有处理好支付，那么支付网关将会产生一个失败的事务，并需要有机制在事后进行补偿处理，从而再次达成数据的一致性。

解决该问题的最好的方式就是对事务管理去中心化。每一个端点应该能做出本地决策从而促成全局范围的事务。我们将在第 3 章讨论更多该主题的内容。

技术对比

当开始构建一个新的软件时，每个开发者的脑海中都应该持有一个概念：标准。

标准能保障你的服务在技术上是独立的，以便于可以方便地将采用不同语言或技术开发的系统进行集成。

采用微服务对系统进行建模的一个优势是可以选择最合适的技术来做相对应的工作，这对于问题的处理来说是相当高效的。而当在构建单块软件的时候，很难像采用微服务那样来组合使用各种技术。通常，在一个单块软件中，我们已经被绑定在最初选择的技术上。

Java 远程方法调用（RMI）就是非标准协议的一个例子，如果你希望系统对新技术是开放的，那么应该避免使用这样的技术。如果所有软件组件都是用 Java 编写的，那么采用这种技术将所有组件连接起来会是一件很棒的事情，但是当开发者使用 Node.js 来调用一个 RMI 方法时将会非常纠结（如果尝试没有失败的话）。这样做会将我们的架构绑死在一个特定的技术上，从微服务的视角来看，这等于扼杀了一个非常有意思的优势：技术多样性。

多微才是足够的微

一旦决定将系统建模成一系列微服务，那么通常首先要回答一个问题：多微才是足够的微？

而答案往往很巧妙，甚至会让人失望：视情况而定。

对于一个特定系统来说，微服务的规模多大才合适取决于公司的组织结构，最好是所创建的软件组件的管理工作正好可以由一个小规模开发团队轻松胜任。这同样也取决于技术需求。

想象有这样一个系统，它专门接收和处理银行文件。你可能也了解，银行之间所有的支付都是以某种已知的特定格式（例如单一欧元支付区，即 SEPA 格式用于欧元支付）的文件来传递的。这类系统的特性之一就是要知道如何处理大量不同的文件。

从微服务的视角来看，解决该问题的首要办法就是将该部分功能从其他所有服务中剥离出来创建成单独的工作单元，并为每一种文件类型创建一个微服务。这样一来，我们便

可以对既有的文件处理器进行修改，而不会妨碍到系统的其他功能。即使在其中某个服务出现问题时，我们仍然可以继续处理其他文件。

微服务的规模应该尽可能小，但是请记住，为了对新的服务进行管理，每个微服务都会给运维团队带来开销。我们来尝试回答一个问题，多微才是足够的微呢？应该从可管理性、可伸缩性以及专职化这几个方面来考虑。微服务应该小到一个人就能承担服务管理工作，并能快速对其扩容（缩容）而不影响系统的其他部分。同时，它应该只做一件事。

常规而言，微服务应该小到足够在一个 sprint 之内完成开发。

关键的好处

在前面的话题中，我们讨论了面向微服务的架构是什么。同时揭示了一些从实战经验中提炼出来的设计原则，并展示了一些这类架构的好处。

现在，我们来看几项关键的好处，并展示一下它们是如何帮助我们提升软件质量并适应新的业务需求的。

弹性

维基百科将弹性定义为*系统处理变化的能力*。我对弹性的理解是在问题被解决后系统从异常状态（短暂的硬件故障以及意料之外的高网络延迟等）或压力期中优雅恢复，同时又不会影响系统性能的能力。

这虽然听上去很简单，但是在构建面向微服务软件的时候，问题源会由于系统的分布式特性而被放大，有时很难（甚至不可能）防止所有的异常情况。

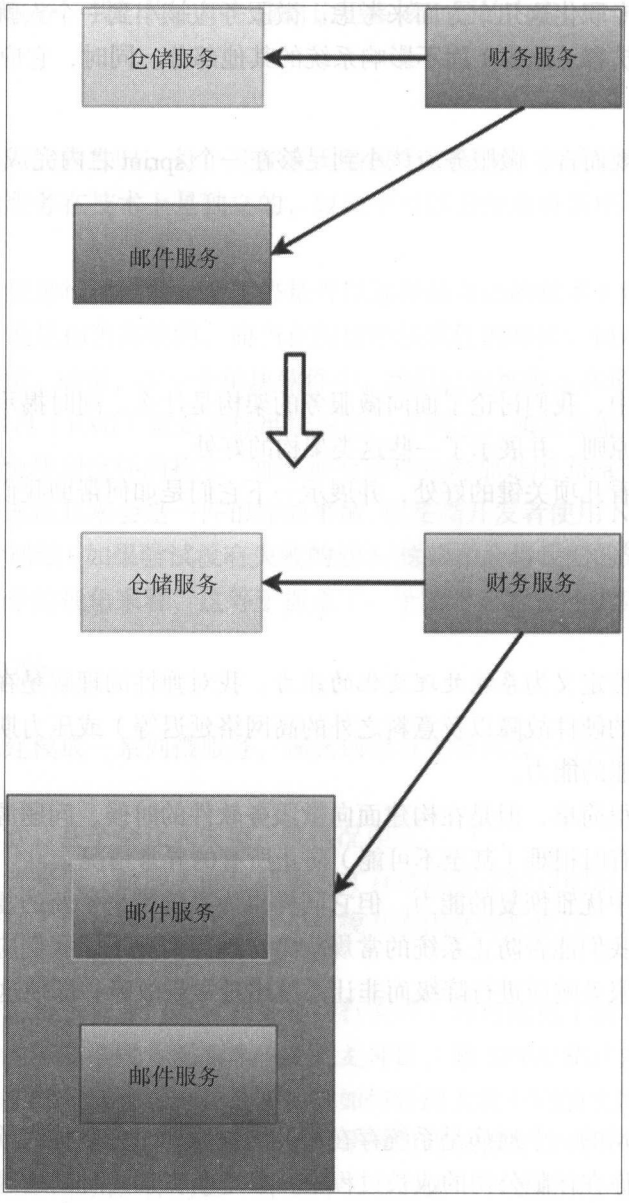
弹性是从错误中优雅恢复的能力。但它同样也为系统带来了新的复杂度：如果一个微服务出现了问题，我们能否防止系统的常规故障？理想情况下，我们应该以这样一种方式来构建服务：仅对服务响应进行降级而非让系统出现常规故障，即使这样做也并不容易。

可伸缩性

如今，各大公司的一个通病是系统存在可伸缩性问题。如果你之前曾与某个单块软件打过交道，我确信你在伴随公司的成长过程中，必定会在某些时刻遭遇到容量问题。

通常，这些问题并不涉及应用的每一层次或所有子系统。往往只有个别子系统或服务会明显慢于其余部分，一旦没有处理好容量问题就会导致整个应用发生故障。

下图描述了我们是如何对微服务进行扩展（扩展成两个邮件服务）的，同时又不牵扯系统的其余部分：



让我们来看一个关于车险的场景，用于计算指定风险因素列表报价的服务便是该类问题的一个例子。通过扩展整个应用来满足对某个特定部分的需求是否有意义？如果你脑海中的答案是“否”，那么你离拥抱微服务更近了一步。微服务可以让你仅仅按需扩展系统的一部分，从而只加大系统特定部分的处理能力。

如果该保险系统是一个面向微服务的系统，那么我们只需要创建更多的微服务实例来负责计算就能解决报价计算需求过旺的问题。请记住，扩展服务会给运维团队增加开销。

技术多样性

软件的世界每几个月就会更新换代。新语言进入业界成为某类系统事实标准的节奏片刻未停。几年前，Ruby on Rails 面世并在 2013 年成为在各种新项目中使用最多的 Web 框架。Golang（由 Google 创建的一门语言）因其结合了强大的性能与优雅简洁的语法而成为当前的一种趋势，任何只要拥有一门编程语言经验的人都可以在几天内学会它。

在过去，我也曾使用 Python 和 Java 成功编写过微服务。

尤其是 Java，自从 Spring Boot 发布之后，它成为在编写敏捷微服务方面相当有吸引力的技术栈。

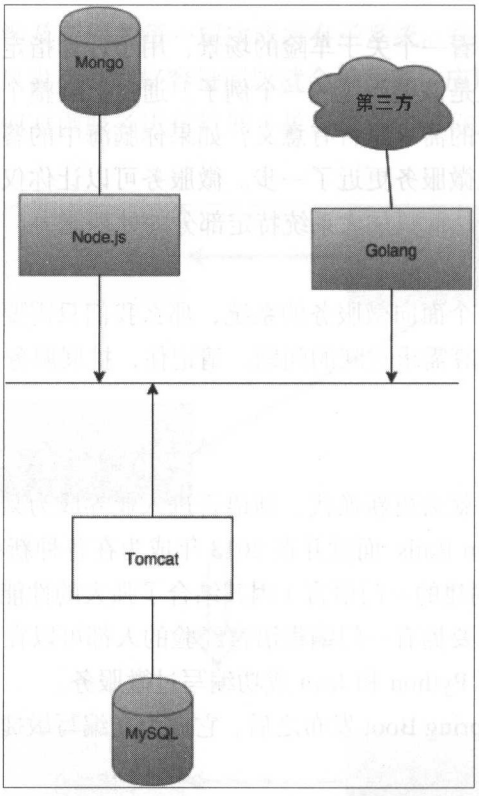
Django 是一款强大且可用于编写微服务的 Python 框架，与 Ruby on Rails 非常相似。通过它我们可以自动化地进行数据库迁移，并可以非常轻松地完成创建 CRUD（创建、读取、更新及删除）服务的工作。

Node.js 利用了著名语言 JavaScript 的优势，创建了一个新的服务端技术栈，从而改变了工程师们编写新软件的方式。

那么，将这些技术都结合起来会有什么问题吗？平心而论，这是一个优势：我们可以选择合适的工具来做相对应的工作。

只要待集成的技术是标准化的，面向微服务的架构便可以帮你实现这一点。正如我们在上文中已经了解到的，一个微服务是非常小的，并且是一个自主运维的软件中的独立部分。

下图展示了微服务是如何隐藏数据的存取逻辑的，两个服务在存取数据方面共用同一个通信点，从而能很好地互相解耦（一个服务实现发生变化时并不涉及任何其他服务）：



此前我们曾讨论到性能的问题。通常，系统的某些部分会比其他部分承受更多的压力。通过利用当代的多核 CPU 进行并行（并发）编程可以解决其中的一些性能问题。然而，Node.js 并不是一门适合执行并行任务的语言。针对那些处于压力之下的微服务来说，我们可以选择一门更加适合的语言来进行开发，比如 Erlang，从而可以以一种更加优雅的方式来管理并发。这样做，花不了你两周的时间。

在同一系统中使用多种技术存在着一个问题：开发人员和系统管理员需要知道所有的（或一部分）相关技能。拥抱微服务的公司通常可以秉持一门核心技术（在本书中，我们将会使用 Node.js），并辅以一些其他技术（我们除了使用 Docker 来管理部署之外，还可以采用 Capistrano 或 Fabricator 来管理发布）。

可替换性

可替换性是指替换系统中某个组件而不影响系统行为的一种能力。

当我们在讨论软件的时候，可替换性往往是与低耦合密不可分的。在编写微服务的时

候不能将内部逻辑暴露给调用服务，服务实现对客户端来说是透明的，客户端了解的只有接口。让我们来看看下面的例子，该接口是用 Java 编写的，仅需通过观察接口就能识别出它存在着什么问题。

```
public interface GeoIpService {
  /**
   * 检查IP是否属于指定ISO代码所对应的国家
   **/
  boolean isIn(String ip, String isoCode) throws
    SOAPFaultException;
}
```

初看该接口可以发现它是自描述的。它将检查特定 IP 是否属于特定的国家，一旦服务出现重大问题会抛出 SOAPFaultException。

如果我们构建客户端来消费该接口，需要考虑到服务的上述逻辑，捕获并处理 SoapFaultException。这等同于将服务内部实现的细节暴露给了外部世界，从而很难再替换掉 GeoIpService 接口。同样的，事实上我们创建的某个服务如果关联了应用逻辑的某个部分则表明创建了一个限界上下文：即一个高内聚的服务或服务集（通过集合所辖服务的协同工作可以达成一个目标）。

独立性

不管我们怎么努力，人类的大脑都不擅长解决复杂问题。人类大脑最有效的运作模式是同一时间只做一件事情，所以我们可以将复杂问题拆解成更小的问题。面向微服务的架构应该也遵从这一方式：所有服务应该都是独立的，它们通过接口进行交互。除了协定确认接口这一环节之外，不同的工程师团队可以在无须交流的情况下完成对服务的开发。一家采用了微服务的公司可以根据业务的需求来调整工程师团队的规模，从而能敏捷地响应业务的高峰期或静默期。

为什么可替换性如此重要

在前面的一个小节中，我们讨论了该如何确定微服务的合理规模。按照普遍的经验而言，一个团队应该能在一个 sprint 内完成一个微服务的重写和部署。这样做的背后的根本原因就是技术债务。

我会将技术债务定义为在一个既定计划的周期内，初始技术设计与预期交付功能之间

的偏差。某些方面的牺牲或错误假设会导致编写的软件非常糟糕，这样的软件需要全盘重构或重写。

在前面的例子中，接口在暴露给外部世界时明确表明必须使用 SOAP 来调用 Web 服务。一旦需要将客户端代码改造成 REST 客户端，REST 客户端根本无法处理 SOAP 异常。

易于部署

微服务应当易于部署。

作为软件开发者，我们知道在软件的部署过程中很多事情都可能会出现问题。

正如前面所提到的，微服务是非常易于部署的，原因如下：

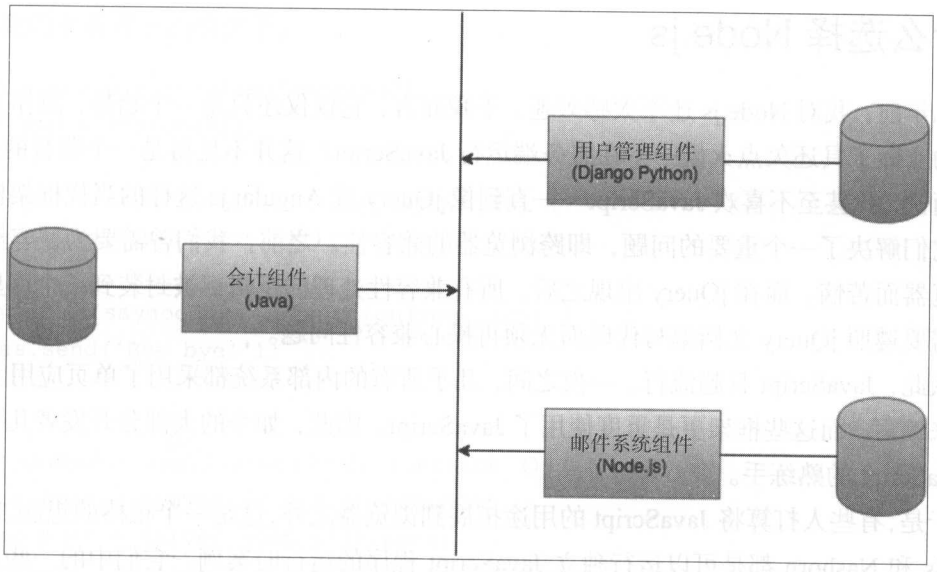
- 少量的业务逻辑（从经验上来说是只需两周即可完成从无到有的编写）导致更易于部署。
- 微服务是自治的工作单元，所以升级一个服务对于复杂系统来说是一个局部可控的问题。无须重新部署整个系统。
- 微服务架构中的基础设施和配置应该尽可能自动化。在本书的后续部分中，我们将学习如何使用 Docker 来部署微服务，以及这样做相比于传统部署技术会有怎样的优势。

SOA 与微服务的比较

面向服务架构（SOA）已经存在有些年头了，这是一种用于设计软件的伟大原则。在 SOA 中，所有组件都是独立自主的，并能为其他组件提供服务。要替换掉系统中的某些部分而不对整个系统造成较大的影响本是个难题，然而只要维护好系统各模块之间的低耦合，该难题便能迎刃而解，这也是我们之前谈及微服务时所认可的。

大体上，SOA 与微服务架构是非常相像的。那么它们之间的区别到底是什么呢？

微服务是细粒度的 SOA 组件。换句话说，某单个 SOA 组件可以被拆成多个微服务，而这些微服务通过分工协作，可以提供与原 SOA 组件相同级别的功能，如下图所示。



微服务是细粒度的 SOA 组件，它们是关注点更窄的轻量级服务。

微服务与 SOA 之间的另一个不同之处是服务互联和编写服务时所使用的技术。

J2EE 是一个遵守企业级标准的用于编写 SOA 架构的技术栈。Java 命名与目录接口（JNDI）、企业级 Java Bean（EJB）以及企业服务总线（ESB）都是 SOA 应用赖以构建和维护的生态土壤。即便 ESB 是标准，在 2005 年之后毕业的工程师却鲜有听说过 ESB 的，至于用过 ESB 的那就更少了。而当代的，例如 Ruby on Rails 这样的框架甚至不会去考虑如此复杂的软件部件。

而另一方面，微服务推崇执行的标准（例如 HTTP）却是人们广泛了解并共同使用的。我们可以通过选择合适的语言或工具来构建某个组件（微服务），进而获得本章“技术多样性”小节所提到的关键好处。

除了技术栈与服务规模之外，在 SOA 与微服务之间还有一个更大的区别：领域模型。在本章前面的内容中，我们曾讨论过去中心化。有管理的去中心化，也有数据的去中心化。在一个基于微服务的软件中，每个微服务应该在本地存储自身管理的数据，并将领域模型分别隔离到单个服务中。而在面向 SOA 的软件中，数据往往存储在单个大型的数据库中，服务之间会共享领域模型。

为什么选择 Node.js

几年前，我对 Node.js 还不太感兴趣。于我而言，它仅仅还只是一个趋势，离作为解决问题的实际工具还欠点火候……在服务端运行 JavaScript？这并不见得是一个明智的选择。平心而论，我甚至不喜欢 JavaScript——直到像 jQuery 或 Angular.js 这样的当代框架横空出世。它们解决了一个重要的问题，即跨浏览器的兼容性。之前，我们曾需要为兼容至少三个浏览器而苦恼。而在 jQuery 出现之后，所有兼容性处理的逻辑都被封装到一个库里，我们只需要遵照 jQuery 文档编写代码而无须再操心兼容性问题。

从此，JavaScript 日趋流行。一夜之间，几乎所有的内部系统都采用了单页应用（SPA）框架来编写，而这些框架更是重度使用了 JavaScript。因此，如今的大部分开发者几乎都成了 JavaScript 的熟练手。

于是，有些人打算将 JavaScript 的用途拓展到浏览器之外，这是一个很棒的想法。Rhino、Node.js 和 Nashorn 都是可以运行独立 JavaScript 程序的运行时案例。它们中的一些运行时还可以与 Java 代码交互，开发者可以向 JavaScript 程序导入 Java 类，从而可以直接复用无数由 Java 编写的框架。

让我们来重点看看 Node.js。Node.js 是用来构建面向微服务的架构的绝佳选择，原因如下：

- 学习门槛低（但是如果想要精通还是有一定门槛的）
- 易于扩展
- 对测试友好
- 易于部署
- 可以通过 npm 进行依赖管理
- 有着大量与主流标准协议相集成的库

基于以上这些原因，结合我们将在后续章节中揭示的其他原因，Node.js 成为构建可靠的微服务的最佳选择。

API 聚合

我选择 Seneca 供后续章节开发之用。Seneca 的一个最吸引人的特性就是 API 聚合。

API 聚合是一项用于将不同功能（插件、方法等）组合成一个接口的高级技术。

让我们来看看下面的例子：

```
var express = require('express');
var app = express();

app.get('/sayhello', function (req, res) {
  res.send('Hello World!');
});
app.get('/saygoodbye', function(req, res) {
  res.send('Bye bye!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('App listening at http://%s:%s', host, port);
});
```

前面的例子使用了 Express，这是一个在 Node.js 技术栈中广为流行的 Web 框架。该框架同样也是围绕 API 聚合技术来构建的。让我们来看看第 4 行和第 7 行。在这些代码里，开发者注册了两个方法。当某人以 GET 请求的方式分别单击 URL：`/sayhello` 和 `/saygoodbye` 时，这两个对应的方法将会被执行。换句话说，该应用由不同的独立且更细粒度的实现组成，然后在单个接口上将这些实现暴露给外部世界。在该例子中，该接口就是一个在 3000 端口上监听的 app。

在下面的章节中，我将解释为什么 API 聚合这一特性是如此重要，而我们又是如何利用它来构建（扩展）微服务的。

展望 Node.js

JavaScript 的设计初衷是成为一门在 Web 浏览器中执行的语言。对于那些在工作或学习中使用 C/C++的人来说，JavaScript 的语法分外亲切。这也是为什么 JavaScript 在 Web 2.0 时代被作为文档动态处理的标准而采用的关键因素。异步 JavaScript 与 XML 技术（AJAX）成为 JavaScript 快速成长的催化剂。然而，不同的浏览器对于 AJAX 请求对象有着各自不同的实现，以至于开发者需要花费大量的时间来编写跨浏览器的代码。

标准的缺失导致出现了大量用于封装 AJAX 背后逻辑的框架，从而帮助我们更容易地

编写跨浏览器的脚本。

JavaScript 是一门脚本语言，它的设计初衷并不是成为一门面向对象语言，同时也不是编写大型应用的最佳选择。往往大型应用的代码会变得越来越混乱，而且很难统一不同公司的代码标准。每个公司各自拥有属于自己的最佳实践，而其中个别公司之间的做法又甚至可能是互相冲突的。

欧洲计算机制造协会（ECMA）开始出来主持大局。ECMAScript 6 是 ECMA 语言（JavaScript、ActionScript、Rhino 等）的下一代标准，它引入了类、继承、集合及其他一些有助于简化 JavaScript 软件开发的有趣特性，相比于 V8 规范显得更加标准。

在这些特性中，我认为最有趣的是引入了 class 关键词，该关键词使我们可以采用对象来构建 JavaScript 软件。

当前，大多数浏览器都支持了大部分这些新特性，但是反观 Node.js，默认只实现了标准的一小部分，而其中一部分的实现还需要我们向解释器传入特殊的标识（harmony flag）才能启用。

在本书中，我将尽可能避免使用 ECMAScript 6 的新特性，并继续使用大部分开发者已经广泛接受的 V8 规范中的内容。一旦你熟悉了 JavaScript V8，将应用迁移到 ECMAScript 6 将是一件轻而易举的事情。

小结

在本章中，我们学习了围绕微服务的一些关键概念，同时也学习了一些设计高质量软件组件的最佳实践。通过遵守这些最佳实践，我们可以构建出健壮及弹性的软件架构，从而能快速响应业务的需求。

你同时也了解了一些面向微服务的架构的关键好处，比如可以为相应的服务选择合适的语言（技术多样性）；以及一些可能会加重我们负担的误区，比如由面向微服务的架构的分布式特性而带来的运维方面的开销。

最后，我们讨论了为什么 Node.js 是用来构建微服务的强大工具，以及如何通过利用像 API 聚合这样的技术来从 JavaScript 获益，从而构建出高质量的软件组件。

在后面的章节中，我们将通过代码示例以及更深入的话题（都是我研究多年的话题）讨论来逐步揭示本章中讨论的这些概念。

正如前面已经说明的，我们将主要专注于 JavaScript 的 V8 版本，但是我也会就如何轻松地编写出可升级的组件从而能拥抱 ECMAScript 6 给出一些提示。

2

基于 Seneca 和 PM2 构建 Node.js 微服务

本章主要介绍两大框架：Seneca 和 PM2，以及它们对于构建微服务的重要性。同时，为了让读者对 Node.js 生态圈有个大体的了解，我们还将介绍一些其他备选方案。本章分为以下三小节。

- **选择 Node.js 的理由**：在本节中，我们将证明选择 Node.js 来构建微服务的正确性。并且，我们还将介绍使用 Node.js 时涉及的软件栈。
- **微服务框架 Seneca**：在本节中，你将学到关于 Seneca 的基本知识，以及它能够使整个软件系统变得易于管理的原因。为了遵循业界标准，我们将教会读者如何整合 Seneca 与 Express（Node.js 平台下最流行的 Web 开发框架）。
- **PM2**：PM2 是运行 Node.js 应用的最好选择。无论你想如何部署应用系统，PM2 都能够提供很好的解决方案。

选择 Node.js 的理由

在之前的章节中，我提到过自己曾经不是一个 Node.js 的追捧者，因为我并不想忍受 JavaScript 标准化程度太低带来的困扰。

最初，JavaScript 只能在浏览器中运行，这是一段痛苦的回忆。在我们使用 JavaScript 时，经常会出现跨浏览器的兼容问题，而且并没有一个统一的标准来解决这个问题。

接着，Node.js 出现了。它具有非阻塞特性（在后续章节中将会进一步介绍），使得我

们能够很容易地创建具有高可伸缩性的应用。而且，由于它是基于 JavaScript 这一风靡已久的语言，因此学习起来也非常容易。

如今，Node.js 已经成为国际上许多科技公司的首选方案。特别的，对于在服务器端需要非阻塞特性（例如 Web Sockets）的场景，Node.js 俨然成了最好的选择。

在本书中，我们将主要使用（但不仅限于）Seneca 和 PM2 作为构建、运行微服务的框架。虽然我们选择了 Seneca 和 PM2，但并不意味着其他框架不好。

业界还存在一些其他备选方案，例如 restify 或 Express 可用于构建应用，forever 或 nodemon 可用于运行应用。然而，我发现 Seneca 和 PM2 是构建微服务的最佳组合。主要原因如下：

- PM2 在应用部署方面有着异常强大的功能。
- Seneca 不仅仅是一个构建微服务的框架，它还是一个范例，能够重塑我们对于面向对象软件的认知。

另外，在本书的一些章节中，我们会结合实例讲述如何将 Seneca 作为中间件与 Express 结合使用。

在讨论这些框架之前，我们准备先介绍一些 Node.js 的概念，这将有助于对框架的理解。

安装 Node.js、npm、Seneca 和 PM2

Node.js 的安装非常容易，根据操作系统的不同，要选择不同的安装包。然后，只需双击安装包，按照指示进行安装即可，安装包会同时安装 Node.js 和 npm（Node Package Manager）。在本书编写时，已经有用于 Windows 和 OS X 系统的安装包。

当然，高级用户（特别是开发运维工程师）可以通过源码或者二进制包来安装 Node.js 和 npm。

可以在 Node.js 的官网上下载到不同平台的安装包，安装包中包括了 Node.js 和 npm（同时也提供各版本的源码、二进制包下载），下载地址：`https://nodejs.org/en/download/`。

Chef 是一款用于搭建服务器环境的配置管理软件，在它的众多特性中，最流行的是 recipe，recipe 可以看作是基于 Chef 进行软件安装、配置的脚本文件。更多说明可以参考以下网址：

`https://github.com/redguide/nodejs`。

在本书编写时，已经有 Linux 平台下的二进制包。

学习 npm

npm 伴随着 Node.js 的出现而产生，它使你可以从互联网上获取依赖包，并且不需要关心这些依赖包的管理问题。通过 npm，你还能够轻易地维护、更新依赖包。同时，也可以通过它来创建全新的工程。

每个 node 应用的根目录下面都有一个 `package.json` 文件，定义了这个项目所需要的各个模块，以及项目的配置信息（比如依赖、版本和常见命令等）。我们看看下面这个例子：

```
{
  "name": "test-project",
  "version": "1.0.0",
  "description": "test project",
  "main": "index.js",
  "scripts": {
  "test": "grunt validate --verbose"
  },
  "author": "David Gonzalez",
  "license": "ISC"
}
```

这是一个自描述文件，文件中值得注意的是 `scripts`。

在本节中，我们可以指定一些特定的命令来执行不同的动作。对于上述例子，如果我们在终端中输入 `npm test`，npm 将执行 `grunt validate --verbose`。

Node 应用一般都是通过执行一条简单的命令来运行的：

```
node index.js
```

不过，你得确保在项目的根节点下存在 `index.js` 文件。一般情况下，最好的方案是在 `package.json` 文件的 `scripts` 中添加如下信息：

```
"scripts": {
  "test": "grunt validate --verbose"
  "start": "node index.js"
},
```

正如你所见，现在我们可以执行两个不同的命令来运行同一个程序：

```
node index.js
npm start
```

使用 npm start 的好处是一致性。无论你的应用多么复杂，都可以使用 npm start 来运行它（只要你的 scripts 配置正确）。

让我们开始在一个全新的工程里安装 Seneca 和 PM2。

首先，登录终端，在安装完 Node.js 之后创建一个新的文件夹，执行 npm init。将会得到与下图类似的提示：

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (newfolder)
```

npm 要求输入一些参数来配置工程，完成这些输入后，它会按照你的输入生成一个 package.json 文件。

接下来，我们需要安装依赖包。npm 可以帮我们完成这项工作，我们只要输入以下命令：

```
npm install --save seneca
```

现在，如果你再次查看 package.json，会发现文件中新增了一个部分——dependencies，其中包括一条关于 Seneca 的依赖描述：

```
"dependencies": {
  "seneca": "^0.7.1"
}
```

这意味着，我们的应用已经能够获取 Seneca 模块，并且 require() 函数也能够找到它。save 标记有多种模式，如下所示。

- save：这种方式会将依赖写入 dependencies 部分，在整个开发周期中，依赖都是可用的。
- save-dev：这种方式会将依赖写入 devDependencies 部分，它只在开发阶段可用，最终不会随着产品一起部署。
- save-optional：这种方式会添加一个依赖（如同 save 一样），但是，如果找不到依赖，它会让 npm 继续运行，交由应用来处理依赖缺失问题。

接下来，继续安装 PM2。虽然 PM2 可以以库的形式使用，但是它主要还是一个命令行工具，类似 UNIX 系统中的 ls、grep。npm 对于安装命令行工具也有很好的支持，可使用如下命令：

```
npm install -g pm2
```

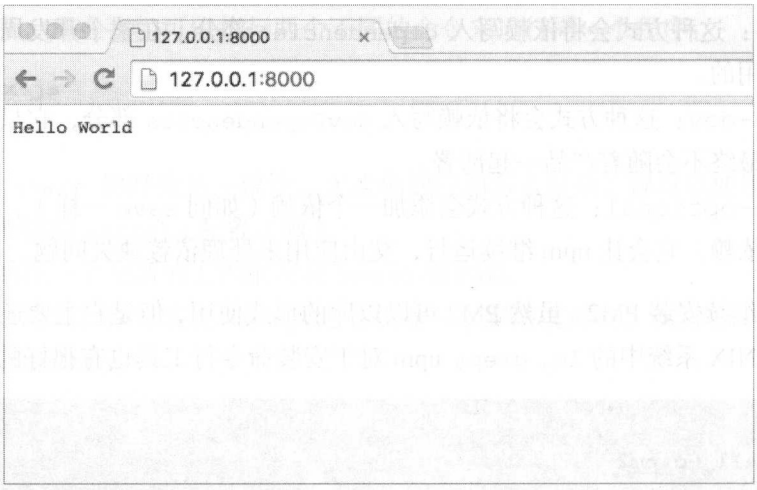
-g 指定 npm 以全局方式安装 PM2，因此它将对整个系统有效，而不仅仅局限于应用中。也就是说，执行上述命令之后，pm2 成为控制台的可执行命令。如果你在终端输入 pm2 help，就能够看到 PM2 的帮助信息。

第一个程序——Hello World

Node.js 中最令人兴奋的理念之一就是简单。只要熟悉 JavaScript，你就可以在几天内学会 Node.js，在几周之内熟练掌握它。用 Node.js 编写的代码要比使用其他语言编写的代码更加简短：

```
var http = require('http');

var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World\n");
});

server.listen(8000);
```

上述代码创建了一个服务端程序，并监听 8000 端口。如若不信，请打开浏览器，在导航栏中输入 http://127.0.0.1:8000，可以看到如下屏幕截图：



让我们来解释一下这段代码：

- 第一行代码载入 `http` 模块。通过调用 `require()` 函数，可以加载 node 的 `http` 模块，并且将该模块的导出分配给变量 `http`。Node.js 正是通过导出的方式将模块的公有方法和属性暴露给外部程序的。
- 第二行代码创建了一个 HTTP 服务。`http` 模块向外界暴露了 `createServer()` 方法。该方法接受一个函数作为入参（注意，函数是 JavaScript 世界中的“一等公民”，它能够作为其他函数的参数进行传递），在 Node.js 中将这种方式称为回调，它是一种响应事件的动作，在本例中，事件表示一个 HTTP 请求。由于 Node.js 是单线程模型，因此它大量地使用了回调函数。Node.js 应用总是运行在一个线程里，因此在等待一个操作完成时采用的是非阻塞的方式，这样可以避免让程序陷入各类等待中。否则，程序不会这么快速地响应请求。当你读到第 4 章时，我们会继续讨论这个问题。
- 在下一行代码中，通过 `server.listen（8000）` 开启服务。从现在开始，每次服务收到请求时，`http.createServer()` 中传入的回调函数将会被执行。

这就是一个完整的服务端程序。“简单”是 Node.js 程序的宗旨。在编写 Node.js 的代码时，你只需要关注核心业务代码，而不需要编写大量复杂的类、方法以及配置对象。如同上例一样，只需要编写一个为请求提供服务的脚本，即可提供 HTTP 服务。

Node.js 的线程模型

在 Node.js 中编写的程序都是单线程的。它的影响非常明显，以上一节中编写的代码为例，如果我们的服务收到上万个并发请求，那么它们将进入等待队列，顺序地被 Node.js 的事件轮询机制处理（这部分内容将在第 4 章与第 6 章中进一步解释）。

从第一感觉来讲，这听起来确实很糟糕。现代 CPU 都是多核的，它们可以并行地处理多个请求。那么 Node.js 采用单线程处理请求有什么好处呢?

答案是：Node.js 采用的是异步处理机制。这表示在处理较慢的事件时，比如读取文件，Node.js 不会阻塞线程，而是继续处理其他事件，Node.js 的控制流在读取文件完毕时，会执行相应的方法来处理返回信息。

仍然以上一节中编写的代码为例，`createServer()`方法接收一个回调函数，这个回调函数将在收到一个 HTTP 请求时被执行。但是在等待 HTTP 请求的同时，线程仍然可以处理其他事件。

但是，这种模型存在一个问题，开发者称之为回调地狱（callback hell）。每个响应阻塞动作的代码都必须以回调的形式存在，这样使代码变得更加复杂。将匿名函数作为`createServer()`方法的参数就是一个很好的例子。

模块化组织的最佳实践

大型项目中的源代码组织方式是一个具有争议的问题。不同的开发者对于如何组织代码采取不同的方法，但是他们都是出于同一个目标——让代码易于管理与阅读。

有些语言，例如 Java、C#，以包（package）的形式组织代码，这样可以在相关包中找到源码文件。举个例子，假如我们正在编写一个任务管理软件，在`com.taskmanager.dao`包中，我们可以寻找到数据访问对象（DAO）的实现类。顾名思义，这些类实现了对于数据库的访问。同样的，在`com.taskmanager.dao.domain.model`包中，我们可以找到所有的模型（通常是表的映射）表示类。

这是 Java、C#中的代码组织惯例。如果你是一名 C#开发者，当开始接触一份已有工程代码，只需要花上几天时间就可以熟悉这份工程代码的组织方式，因为语言惯例强制约定了代码组织方式。

JavaScript

JavaScript 最初设计成只能运行在浏览器中，它可以被嵌入 HTML 文档，通过操作 DOM

元素产生动态效果。让我们看以下例子：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Title of the document</title>
</head>
<body>
  Hello <span id="world">Mundo</span>
  <script type="text/javascript">
  document.getElementById("world").innerText = 'World';
  </script>
</body>
</html>
```

如你所见，如果你在浏览器中加载这段 HTML，id 为“world”的 span 标签中的内容将被替换成“World”。

在 JavaScript 中，并没有依赖管理这一概念。JavaScript 能从 HTML 中分离出来，放入它自己的文件中。但是，并不能（至少现在是）将一个 JavaScript 文件引入到其他 JavaScript 文件中去。

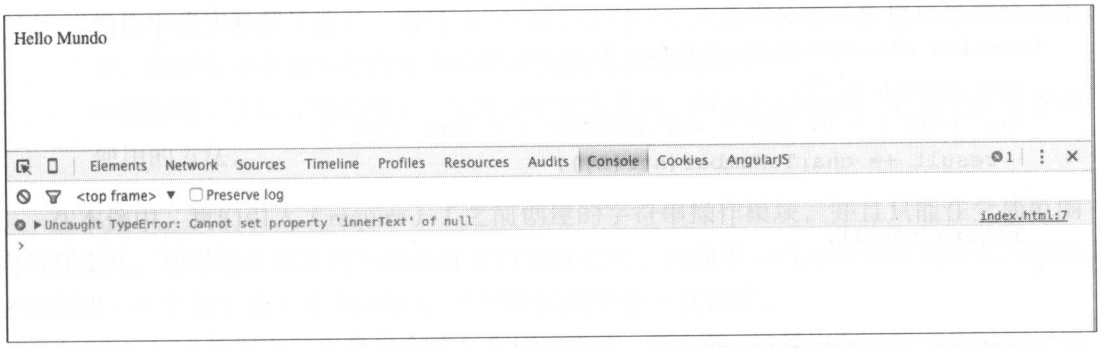
这导致了一个很大的问题。当工程代码包含了许多 JavaScript 文件时，管理文件更像是一门艺术，而不是工程工作。

因为浏览器会在第一次发现 JavaScript 文件时便立即执行它们，所以导入 JavaScript 文件的顺序变得相当重要。让我们修改前例中的代码顺序来说明这个问题，如下所示：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Title of the document</title>
  <script type="text/javascript">
    document.getElementById("world").innerText = 'World';
  </script>
</head>
<body>
  Hello <span id="world">Mundo</span>
```

```
</body>
</html>
```

现在，将 HTML 文件保存在 `index.html` 文件中，然后使用任一浏览器来加载这个文件，会出现如下所示的结果：



在本例中，通过 Chrome 开发者工具中的控制台，可以看见以下错误：Uncaught TypeError: Cannot set property 'innerText' of null。

为什么会发生这种情况？

正如我们之前提到过的，浏览器按顺序执行代码，当浏览器执行到这段 JavaScript 代码时，`world` 元素还不存在。

幸运的是，Node.js 已经使用一种很优雅和标准的方式解决了依赖加载问题。

SOLID 设计原则

每当谈论到微服务，我们总会提及模块化，而模块化归结于以下设计原则：

- 单一职责原则
- 开放封闭原则（对扩展开放，对修改关闭）
- 里氏替换原则
- 接口分离原则
- 依赖倒置原则（反转控制和依赖注入）

你应该将代码以模块的形式进行组织。一个模块应该是代码的聚合，它负责简单地处理某件事，并且可以处理得很好，例如操作字符串。但是请注意，你的模块包含越多的函数（类、工具），它将越缺乏内聚性，这是应该极力避免的。

在 Node.js 中，每个 JavaScript 文件默认是一个模块。当然，也可以使用文件夹的形式组织模块，但是我们现在只关注使用文件的形式：

```
function contains(a, b) {
  return a.indexOf(b) > -1;
}

function stringToOrdinal(str) {
  var result = ""
  for (var i = 0, len = str.length; i < len; i++) {
    result += charToNumber(str[i]);
  }
  return result;
}

function charToNumber(char) {
  return char.charCodeAt(0) - 96;
}

module.exports = {
  contains: contains,
  stringToOrdinal: stringToOrdinal
}
```

以上代码是一个有效的 Node.js 模块。这个模块有三个函数，其中两个作为公有函数暴露给外部模块使用。

在 Node.js 中可以通过 `module.exports` 变量实现以上功能。只有分配给这个变量的对象，才能对模块外部环境可见，因此可以在模块中实现私有域，例如本例中的 `charToNumber()` 函数一样。

如果想使用这个模块，只需调用 `require()` 函数，如下所示：

```
var stringManipulation = require("./string-manipulation");
console.log(stringManipulation.stringToOrdinal("aabb"));
```

输出结果是 `1122`。

结合 SOLID 设计原则，回顾一下我们的模块。

- **单一职责原则**：模块只处理字符串。
- **开放封闭原则（对扩展开放，对修改关闭）**：可以为模块添加更多的函数，那些已有的正确函数可用于构建模块中的新函数，同时，我们不对公用代码进行修改。
- **里氏替换原则**：跳过这个原则，因为该模块的结构并没有体现这一原则。
- **接口分离原则**：JavaScript 与 Java、C#不同，它不是一门面向接口的语言。但是本模块中确实暴露了接口。通过 `module.exports` 变量将公有函数接口暴露给调用者，这样具体实现的修改并不会影响到调用者的代码编写。
- **依赖倒置**：这是失败的地方，虽然不是彻底失败，但也足以使我们必须重新考量所使用的方法。

在本例中，我们引入（require）了之前创建的字符串操作模块，并且只能在全局范围内与其交互。如果想在模块内与模块外进行数据交互，目前唯一可行的方法是在引入模块之前创建一个全局变量（或者函数），并且保证该变量一直有效。

在 Node.js 中使用全局变量是个让人头痛的事情。请注意，如果你在声明变量时忘记加上 `var` 关键字，它自动会成为全局变量。

使用全局变量会将各个模块耦合在一起，但是过度耦合是我们无论如何都要避免的。因此，应该寻找一个更好的方法来定义微服务中的各个模块（在其他 JavaScript 工程中也应避免耦合）。

我们重构上例代码：

```
function init(options) {

  function charToNumber(char) {
    return char.charCodeAt(0) - 96;
  }

  function StringManipulation() {
  }

  var stringManipulation = new StringManipulation();

  stringManipulation.contains = function(a, b) {
    return a.indexOf(b) > -1;
  };
```

```
  stringManipulation.stringToOrdinal = function(str) {
    var result = ""
    for (var i = 0, len = str.length; i < len; i++) {
      result += charToNumber(str[i]);
    }
    return result;
  }
  return stringManipulation;
}

module.exports = init;
```

这看起来比重构前的代码更加复杂了，但是一旦你习惯了这种方式，将受益匪浅：

- 我们能够向模块内部传递配置参数（例如调试信息）。
- 如果所有成员都内聚在函数内部，那么可以避免它们被全局变量污染。此外，可以使用 use strict，加入这个声明后，如果声明变量时没有带上 var，则会报编译错误，而不是自动转为全局变量。
- 将模块参数化使得测试时能够轻易地 mock 行为与数据。

在本书中，我们将从微服务的角度编写大量代码来构建系统。同时，我们会尽最大可能地使用以上模式编写代码，这将进一步展示该模式带来的好处。

我们即将使用的微服务库 Seneca 也遵循这个模式，同样的，网上许多 JavaScript 库都采用了相同的模式。

微服务框架 Seneca

Seneca 是一个用于构建微服务的框架，它的作者是 nearForm 公司的创始人兼 CTO：Richard Rodger，nearForm 是使用 Node.js 为其他公司提供软件设计与实现的咨询公司。Seneca 相当简单，它使用完备的模式匹配接口来连接各个服务，从代码中将数据传输抽象出来，使编写具有高可扩展性的软件变得相当容易。

让我们直接看以下例子：

```
var seneca = require( 'seneca' )()

seneca.add({role: 'math', cmd: 'sum'}, function (msg, respond) {
```

```
  var sum = msg.left + msg.right
  respond(null, {answer: sum})
})

seneca.add({role: 'math', cmd: 'product'}, function (msg, respond) {
  var product = msg.left * msg.right
  respond( null, { answer: product } )
})

seneca.act({role: 'math', cmd: 'sum', left: 1, right: 2},
  console.log)
    seneca.act({role: 'math', cmd: 'product', left: 3, right: 4},
  console.log)
```

如你所见，代码的意思一目了然：

- Seneca 本身是一个模块，因此首先需要通过 `require()`获取该模块，接着调用 Seneca 的包装函数完成代码库的初始化。
- 接下来的两条命令和第 1 章中提到的“API 聚合” 概念有关。`seneca.add()`方法可以为 Seneca 添加能在特定模式下被调用的函数。这里，我们定义了两个函数，第一个在 Seneca 接收到`{role:math, cmd:sum}`命令时被调用，第二个在接收到`{role: math, cmd: product}`时被调用。
- 在最后一行中，act 函数的第一个入参表示相应的命令，能够触发 Seneca 调用与其匹配的服务。例如，第一个 act 中的参数能够匹配第一个服务，第二个 act 中的参数能够匹配第二个服务。

在之前创建的 `index.js` 文件中编写上述代码（前提是已经安装了 Seneca 和 PM2），并且执行如下命令：

```
node index.js
```

将会得到类似下图的输出结果：

```
→ code  node index.js
2016-03-07T20:28:20.636Z 3xdlpxcs8rjk/1457382500629/2233/- INFO hello   Seneca/1.3.0/3xdlpxcs8rjk/1457382500629/2233/-
null { answer: 3 }
null { answer: 12 }
```

随后，我们将解释这个结果的含义。如果你熟悉企业级应用，大概已经能猜到发生了

什么。

最后的两行是服务的返回结果，第一个执行了 1+2，第二个执行了 3*4。

这两个返回结果都以 null 开头，这在 JavaScript 中是一个很常用的模式——错误优先回调。

用下例来解释这个模式：

```
var seneca = require( 'seneca' )()

seneca.add({role: 'math', cmd: 'sum'}, function (msg, respond) {
  var sum = msg.left + msg.right
  respond(null, {answer: sum})
})

seneca.add({role: 'math', cmd: 'product'}, function (msg, respond) {
  var product = msg.left * msg.right
  respond( null, { answer: product } )
})

seneca.act({role: 'math', cmd: 'sum', left: 1, right: 2},
  function(err, data) {
  if (err) {
    return console.error(err);
  }
  console.log(data);
});
seneca.act({role: 'math', cmd: 'product', left: 3, right: 4},
  console.log);
```

上面这段代码以更加合理的方式重写了第一个调用 Seneca 的方法。这里，并不是将所有东西都打印到控制台，而是先处理 response。回调函数中第一个参数如果为 error（非 null）则打印错误信息，第二个参数是从微服务中返回的数据。这就是为什么在第一个例子中，每行行首都打印为 null。

在 Node.js 的世界中，使用回调是非常普遍的。回调提供了一种动态响应事件的方法，避免了程序阻塞地等待前一个任务的完成。Seneca 也不例外，它大量地使用回调来处理服务调用的返回结果，这对于在不同机器上部署的微服务更为重要（在之前的例子中，所有服务都运行在同一台机器上），因为此时网络延迟将成为软件设计时需要考虑的要素。

实现控制反转

控制反转思想在现代软件中是不可或缺的，随之而来的还有依赖注入。

控制反转是一种软件思想，它能代理创建或调用各组件及方法，使得模块本身不用关注创建它们所需要的依赖，这些通常是通过依赖注入完成的。

Seneca 并没有使用依赖注入，但是它是实现控制反转思想的典型例子。

看一下以下代码：

```
var seneca = require('seneca')();
seneca.add({component: 'greeter'}, function(msg, respond) {
  respond(null, {message: 'Hello ' + msg.name});
});
seneca.act({component: 'greeter', name: 'David'}, function(error,
  response) {
  if(error) return console.log(error);
  console.log(response.message);
});
```

这是最基本的 Seneca 示例。从企业级软件的角度出发，我们可以从中区分出两个组件：生产者（`Seneca.add()`）和消费者（`Seneca.act()`）。如之前提到的，Seneca 没有使用依赖注入，但却很优雅地依据控制反转原则构建了它的代码。

在 `Seneca.act()` 函数中，我们并没有显式地调用处理业务逻辑的组件，而是通过 JSON 信息向 Seneca 指定具体调用的组件。这就是控制反转。

Seneca 在处理控制反转上相当灵活，没有关键字和强制的字段。它只需一组键值对，被用于模式匹配引擎 Patrun 中。

Seneca 的模式匹配

模式匹配是微服务中最灵活的软件模式之一。

与网络地址或信息不同，模式很容易扩展。通过下例来解释这个观点：

```
var seneca = require('seneca')();
seneca.add({cmd: 'wordcount'}, function(msg, respond) {
  var length = msg.phrase.split(' ').length;
  respond(null, {words: length});
});
```

```
seneca.act({cmd: 'wordcount', phrase: 'Hello world this is
  Seneca'}, function(err, response) {
  console.log(response);
});
```

这是一个用于统计句子中单词数量的服务。可以看到，通过 seneca.add()函数，我们为 wordcount 命令添加了处理器，并且在第二句调用中向 Seneca 发送了统计短语中单词个数的请求。

如果执行以上代码，将会得到与下图相似的结果：

```
→ code  node string-require.js
1122
```

到这里，你应该能够明白它是如何工作的了，甚至能够对其做出改动。

现在，让我们对其进行扩展，统计时跳过较短的单词（长度小于等于 3），如下所示：

```
var seneca = require('seneca')();

seneca.add({cmd: 'wordcount'}, function(msg, respond) {
  var length = msg.phrase.split(' ').length;
  respond(null, {words: length});
});

seneca.add({cmd: 'wordcount', skipShort: true}, function(msg,
  respond) {
  var words = msg.phrase.split(' ');
  var validWords = 0;
  for (var i = 0; i < words.length; i++) {
    if (words[i].length > 3) {
      validWords++;
    }
  }
  respond(null, {words: validWords});
});

seneca.act({cmd: 'wordcount', phrase: 'Hello world this is
  Seneca'}, function(err, response) {
```

```
    console.log(response);
});

seneca.act({cmd: 'wordcount', skipShort: true, phrase: 'Hello
  world this is Seneca'}, function(err, response) {
  console.log(response);
});
```

如你所见，我们为 `wordcount` 命令添加了另一个处理器，并添加了一个额外的参数 `skipShort`。

这个处理器在统计单词数时跳过了长度小于等于 3 的单词，执行上述代码，将得到类似下图所示的输出：

```
➜ code  node wordcount.js
2015-11-01T13:50:05.889Z hrzpzs2mgt2n/1446385805876/3897/- INFO hello   Seneca/0.7.2/hrzpzs2mgt2n/1446385805876/3897/-
{ words: 5 }
{ words: 4 }
```

第一行的`{words:5}`是调用第一个 act 的结果，第二行的`{words:4}`是调用第二个 act 的结果。

模式匹配库 Patrun

Patrun 同样是由 Richard Rodger 编写的，Seneca 使用它来执行模式匹配，以判断该由哪一个服务来响应请求。

Patrun 使用最近匹配原则来处理调用。让我们对下例进行分析：

```
{ x:1,     } -> A
{ x:1, y:1 } -> B
{ x:1, y:2 } -> C
```

在上图中，我们可以看到 3 种模式。这与上例中 `seneca.add()` 函数中的模式相同。

在本例中，我们注册了三种关于 x、y 不同取值的组合，来看看 Patrun 是如何对它们进行匹配的：

- `{x: 1} ->A`: 这与 A 完全匹配。
- `{x: 2} ->`: 无匹配项。

- `{x:1, y:1} -> B`：与 B 完全匹配；虽然它与 A 也能匹配，但是显然与 B 的匹配度更高——两个匹配项与一个匹配项的区别。
- `{x:1, y:2} -> C`：与 C 完全匹配，同理，与 A 也匹配，但是 C 的匹配度更高。
- `{y: 1} ->`：无匹配项

如你所见，Patrun（在 Seneca 中）总是获取最长匹配项。因此，我们能够轻易地通过具象化匹配来扩展出更多抽象模式的功能。

复用模式

在前文的例子中，为了实现跳过小于等于三个字母的单词统计功能时，我们重新编写了一个函数，并没有复用之前基础的单词统计函数。

因为，在这个例子中，复用函数是相当困难的。虽然功能需求看起来非常类似，但是解决方案上几乎没有重叠的部分。

然而，我们可以回到两数相加的例子，通过这个例子能体现函数复用：

```
var seneca = require( 'seneca' )()

seneca.add({role: 'math', cmd: 'sum'}, function (msg, respond) {
  var sum = msg.left + msg.right
  respond(null, {answer: sum})
});

seneca.add({role: 'math', cmd: 'sum', integer: true}, function
  (msg, respond) {
  this.act({role: 'math', cmd: 'sum', left: Math.floor(msg.left),
    right: Math.floor(msg.right)},respond);
});

seneca.act({role: 'math', cmd: 'sum', left: 1.5, right: 2.5},
  console.log)

seneca.act({role: 'math', cmd: 'sum', left: 1.5, right: 2.5,
  integer: true}, console.log)
```

如你所见，代码只是稍加改变。接收 integer 的模式依赖基础模式计算两数之和。Patrun 从以下两个角度来匹配最接近、最具体的模式：

- 最长的匹配链
- 模式中元素的顺序

Patrun 会搜寻最优匹配结果，如果存在多个匹配度相同的最优解，那么将匹配第一个结果。

通过这种方式，我们可以依赖于已存在的模式来构建新的服务。

编写插件

插件是 Seneca 应用的重要组成部分之一。正如第 1 章中提到的，在微服务架构中，通过 API 聚合来构建应用是一种很好的方式。

Node.js 的众多流行框架都遵循以下原则：由一系列小软件的组合来构建一个更大的系统。

Seneca 也是依照这个原则来构建的。通过 `Seneca.add()` 函数来为新问题添加处理模块，因此，最终的 API 将会是一系列小的软件模块的组合。

此外，Seneca 还进一步实现了一套令人兴奋的插件系统，通用的功能可以被模块化并抽象成可复用的组件。

下面这个例子是 Seneca 中的一个小插件：

```
function minimal_plugin( options ) {
  console.log(options)
}

require( 'seneca' )()
  .use( minimal_plugin, {foo:'bar'} )
```

将以上代码写入 `minimal-plugins.js` 文件中并执行：

```
node minimal-plugin.js
```

你将得到与下图类似的结果：

```
➜ code node minimal-plugin.js
2016-04-10T22:22:14.849Z lojwswfluxej/1460326934841/6893/- INFO hello   Seneca/1
.3.0/lojwswfluxej/1460326934841/6893/-
{ foo: 'bar' }
```

在 Seneca 中，插件在启动时被加载，由于默认的日志级别为 INFO，而插件加载的日

志级别为 DEBUG，因此默认情况下我们无法看到插件加载信息。但是，可以通过添加参数来获取更多日志信息，如下所示：

```
node minimal-plugin.js --seneca.log.all
```

这时，你将会得到大量的输出，因为输出内容几乎包括了 Seneca 内部运行的全部信息。这些信息对于我们调试复杂场景非常有用，但此时我们只需要显示插件列表：

```
node minimal-plugin.js --seneca.log.all | grep plugin | grep DEFINE
```

执行以上命令将得到与下图类似的结果：

```
2015-11-01T20:43:38.969Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  basic         DEFINE  {}
2015-11-01T20:43:39.230Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  transport DEFINE  {}
2015-11-01T20:43:39.388Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  web         DEFINE  {}
2015-11-01T20:43:39.420Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  mem-store DEFINE  {}
2015-11-01T20:43:39.425Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  minimal_plugin  DEFINE  {foo:bar}
```

让我们分析一下以上输出。

- `basic`：该插件包含在 Seneca 的主模块中，提供一系列基础且实用的 action 模式。
- `transport`：传输插件。直到现在，我们只是在同一台机器上执行不同的服务（相当微小且简单），如果想要将它们分发部署该怎么做？这个插件能够提供帮助，在后面的章节中将学习如何使用它。
- `web`：在第 1 章中，我们提到微服务应该采用广泛使用的标准管道作为它们的连接方式。Seneca 默认使用 TCP 协议，创建 RESTful API 相当麻烦。这个插件可帮助我们更好地编写 RESTful API，我们将在后续章节中学习它的使用方法。
- `mem-store`：Seneca 提供了数据抽象层，因此可以使用不同的底层存储，例如 Mongo、SQL 类数据库等。Seneca 通过 mem-store 提供了让我们开箱即用的内存存储功能。
- `minimal_plugin`:这是我们创建的插件。现在我们知道 Seneca 已经能够加载它了。

我们编写的插件并没有实际功能，现在，让我们来编写有实际作用的代码：

```
function math( options ) {

  this.add({role:'math', cmd: 'sum'}, function( msg, respond ) {
    respond( null, { answer: msg.left + msg.right } )
  })
```

```
  this.add({role:'math', cmd: 'product'}, function( msg, respond )
    {
    respond( null, { answer: msg.left * msg.right } )
  })

}

require( 'seneca' )()
  .use( math )
  .act( 'role:math,cmd:sum,left:1,right:2', console.log )
```

首先，注意最后一条命令，act()中使用了一种不同的样式。我们没有传入字典，而是传入了与字典中键值对内容相同的一个字符串。这并没有问题，但是我个人偏向于使用JSON对象(字典)作为入参，因为通过这种方式组织数据可以避免一些不必要的语法问题。

在之前的例子中，我们已经学会如何以插件的形式组织代码。如果执行这个文件，将得到与下图所示类似的输出：

```
➜ code node math.js
2016-03-07T20:39:33.145Z kw4uoq06n1xg/1457383173137/2623/- INFO hello   Seneca/1.3.0/kw4uoq06n1xg/1457383173137/2623/-
null { answer: 3 }
```

在使用Seneca时，如何初始化插件是一个值得注意的问题。插件的包装函数(上例中的 math()函数)被称为定义函数，在设计上是同步执行的。之前我们提到过，Node.js 应用是单线程运行的。

在初始化插件时，应该添加一个特定的 init() action 模式。每个插件的初始化动作将会顺序执行。init()函数必须无误地调用 respond()回调函数，如果插件初始化失败，Seneca 将退出 Node.js 进程。当然，你肯定也希望微服务在遇到问题时能快速失败并且报错。注意，在执行任何 action 前，必须保证所有插件都成功初始化。

让我们一起看看以下方法如何初始化插件：

```
function init(msg, respond) {
  console.log("plugin initialized!");
  console.log("expensive operation taking place now... DONE!");
  respond();
}

function math( options ) {
```

```
    this.add({role:'math', cmd: 'sum'}, function( msg, respond ) {
      respond( null, { answer: msg.left + msg.right } )
    })

    this.add({role:'math', cmd: 'product'}, function( msg, respond )
      {
      respond( null, { answer: msg.left * msg.right } )
    })

    this.add({init: "math"}, init);
  }
  require( 'seneca' )()
    .use( math )
    .act( 'role:math,cmd:sum,left:1,right:2', console.log )
```

执行上述代码后，将得到与下图所示类似的输出：

```
➜ code  node expensive.js
2016-03-07T20:40:25.351Z bv8phjhz4b92/1457383225343/2640/- INFO hello   Seneca/1.3.0/bv8phjhz4b92/1457383225343/2640/-
plugin initialized!
expensive operation taking place now... DONE!
null { answer: 3 }
```

从输出中可以看出，插件的初始化函数已经被成功调用。

Node.js 应用的一个原则是从不阻塞线程。如果你发现阻塞线程，应该想想如何避免它。

整合 Web 服务器

在第 1 章中特别强调过，我们使用标准技术来进行微服务之间的通信。

Seneca 默认通过 TCP 传输层进行信息交互。虽然它使用 TCP，但是与之交互却并不容易，因为最终执行响应的方法是由客户端发送的请求决定的。

让我们讨论一个更普遍的用例：服务的调用方是浏览器中的 JavaScript。虽然通过普通 JSON 会话就能满足需求，但是如果 Seneca 提供 REST API 来代替它会更简单。这对于微服务之间的通信来说是完美的选择，除非你要求极低的延迟。

Seneca 并不是一个 Web 框架。它被定义为一个通用的微服务框架，因此它并不会对具体的某个应用场景（例如 Web）做过多的支持。

取而代之的是，Seneca 能够非常轻易地与其他框架进行整合。

Express 是基于 Node.js 构建 Web 应用的首选。在网络上可以找到大量关于 Express 的例子与文档，因此学习使用它非常容易。

将 Seneca 作为 Express 的中间件

Express 也是基于 API 聚合原则构建的。在 Express 中，每个软件模块都被称为中间件，它们在代码中以链式结构串联，以此来处理每个请求。

我们准备将 seneca-web 作为 Express 的一个中间件，因此只要指定了配置，所有的 URL 都将遵循规定的命名规范。

请看以下例子：

```
var seneca = require('seneca')()

seneca.add('role:api,cmd:bazinga',function(args,done){
  done(null,{bar:"Bazinga!"});
});
seneca.act('role:web',{use:{
  prefix: '/my-api',
  pin: {role:'api',cmd:'*'},

  map:{
    bazinga: {GET: true}
  }
}})
var express = require('express')
var app = express()
app.use( seneca.export('web') )
app.listen(3000)
```

这段代码并不像之前的例子那么好理解，但是我会尽最大的努力去解释它：

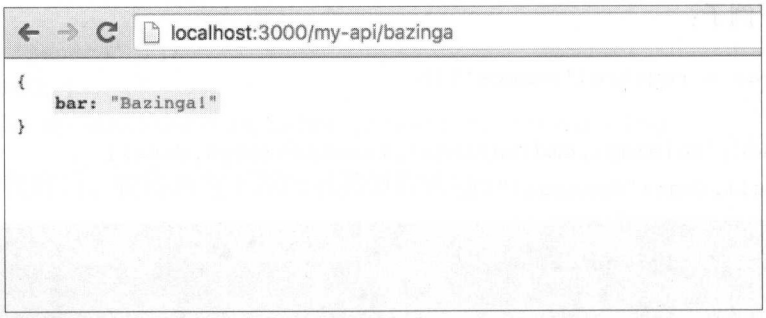
- 第二行代码为 Seneca 添加了一个模式。你应该对这行代码非常熟悉，因为本书中所有的例子都是这么做的。
- 请关注第三条命令 `seneca.act()`，这正是本例中最神奇的地方。我们将 `role:api` 模式与任意的 cmd 模式（`cmd:*`）装配到一起，以响应 `/my-api` 下的 URL 请求。在本例中，第一个 `seneca.add()` 将响应 URL `/my-api/bazinga` 的请求，因为在

seneca.act()中，prefix变量被定义为/my-api，并且seneca.add()中cmd模式下指定了bazinga。

- app.use(seneca.export('web'))指定seneca-web作为Express的中间件，并根据配置规则执行相关动作。
- app.listen(3000)将Express与3000端口进行绑定。

本章之前提到过，seneca.act()将一个函数作为第二个参数。在本例中，我们将请求与Seneca响应动作的映射关系作为配置传递给Express。

让我们测试一下上述代码：



上述代码确实有些晦涩，我们将从浏览器到代码的角度进行解释。

- Express收到请求之后，将其交付给seneca-web处理。
- 所有以/my-api为前缀的请求，都会路由到seneca-web进行处理。在以上代码的seneca.act()函数中，通过关键字pin将role:api模式和任意cmd模式（cmd:*）绑定到seneca-web的响应action上。通过这种方式，/my-api/bazinga与第一个seneca.add()中添加的{role:'api', cmd: 'bazinga'}模式绑定在了一起。

想要完全掌握如何结合Seneca与Express需要耗费一点工夫，但是如果你能完全掌握，API聚合模式带来的灵活性是无限的。

Express本身非常庞大，已经超出了本书的范畴。但是由于它是一个非常流行的Web框架，因此值得我们对其进行简短介绍。

数据存储

Seneca具有数据抽象层，允许我们使用通用的方式操作应用的数据。

正如本章之前介绍过的，Seneca默认加载in-memory存储插件，因此，我们可以直接

使用它。

在本书接下来的章节中，我们将会大量使用到它。底层存储系统各式各样，已超出了本书范畴，但是，Seneca 对它们进行了抽象。

Seneca 基于以下操作，提供了简单抽象数据层（ORM，对象关系映射）。

- load：通过标识符读取实体。
- save：创建实体或者通过标识符更新实体。
- list：列出满足查询条件的所有实体。
- remove：删除指定标识符对应的实体。

让我们来构建一个管理数据库中员工信息的插件：

```
module.exports = function(options) {
  this.add({role: 'employee', cmd: 'add'}, function(msg, respond){
    this.make('employee').data$(msg.data).save$(respond);
  });

  this.find({role: 'employee', cmd: 'get'}, function(msg, respond)
    {
    this.make('employee').load$(msg.id, respond);
  });
}
```

记住一点，由于我们默认使用内存数据库，因此现在不需要关心表结构。

第一条命令向数据库中添加一条员工信息。第二条命令通过 id 从数据中获取一位员工信息。

注意，Seneca 中所有的 ORM 原语都是以$结尾的。

如你所见，现在我们已经从具体数据存储实例中抽象出来了。如果有一天，应用发生变更，必须使用 MongoDB 替代内存存储，我们唯一需要关注的是 MongoDB 的相关插件。

我们将使用员工管理插件，代码如下所示：

```
var seneca = require('seneca')().use('employees-storage')
var employee = {
  name: "David",
  surname: "Gonzalez",
  position: "Software Developer"
}
```

```
function add_employee() {
  seneca.act({role: 'employee', cmd: 'add', data: employee},
    function (err, msg) {
    console.log(msg);
  });
}
add_employee();
```

上例中，我们在代码中使用了存储插件，通过模式匹配将员工信息存入内存数据库中。

在本书中，我们将看到使用数据抽象层的不同例子。但是，主要关注点是如何构建微服务，而不是如何处理各种不同的数据存储。

PM2——Node.js 的任务执行器

PM2 是一款可以为服务器实例带来负载均衡功能的生产级别的进程管理器，通过 PM2 我们可以自由伸缩 Node.js 应用。此外，它能确保进程持续运行，解决 Node.js 单线程模型带来的副作用：一个没有被捕获的异常通过杀死线程，进而杀死整个应用。

单线程应用及异常

前面我们提到过，Node.js 应用是单线程执行的，这不表示 Node.js 不能并发。它表示你的应用是以单线程模式运行的，而其余任务是并行的。

单线程模式意味着，*如果抛出的异常没有被处理的话，应用程序将会挂掉。*

这个问题可以通过使用 promise 库（例如 bluebird）解决；通过 promise 方式，应用不仅可以处理成功的返回，还能够处理异常，因此它可以防止异常“冒泡”导致应用崩溃。

然而，还是存在一些在我们控制范围之外的情况，我们称之为*不可恢复的错误*。一旦出现这些错误，最终将导致你的应用程序崩溃。在诸如 Java 的语言中，异常并不是什么大问题，这些异常虽然导致线程的死亡，但是应用仍然会继续工作。

在 Node.js 中，这却是一个大问题。但是，我们可以通过任务执行器，例如 forever，来解决这个问题。

forever 与 PM2 都是任务执行器，当你的应用意外退出时，它们可以重启你的应用，从而能确保其正常运行。

请看以下例子：

```
➜ ~ forever helloWorld.js
warn:    --minUptime not set. Defaulting to: 1000ms
warn:    --spinSleepTime not set. Your script will exit if it does not stay up for at least 1000ms
Server running at http://127.0.0.1:8000/
```

当 `hellorWorld.js` 被 forever 控制时，forever 会在 helloWorld 应用挂掉之后，重启应用。如果我们杀死 helloWorld 应用，可以得到如下所示的结果：

```
4902 ttys000    0:00.33 node /usr/local/bin/forever helloWorld.js
4903 ttys000    0:00.08 /usr/local/bin/node /Users/dgonzalez/helloWorld.js
```

如你所见，forever 生成另外一个 PID 为 `4903` 的进程。现在，通过 kill 命令杀死该进程（`kill -9 4903`），以下是 forever 的输出结果：

```
➜ ~ forever helloWorld.js
warn:    --minUptime not set. Defaulting to: 1000ms
warn:    --spinSleepTime not set. Your script will exit if it does not stay up for at least 1000ms
Server running at http://127.0.0.1:8000/
error: Forever detected script was killed by signal: SIGKILL
error: Script restart attempt #1
Server running at http://127.0.0.1:8000/
```

虽然我们已经杀死了该进程，但是 forever 会立刻重启应用，几乎不会出现可感知的停机时间。

如你所见，forever 相当原始：无论你杀死应用多少次，它都会将其重启。

在 Node.js 生态圈中，还有一个相当有用的工具包 nodemon。当它探测到监控的文件（默认监控工程下所有文件，即`*.*`）发生变化时，它将重载应用。

```
➜ ~ nodemon helloWorld.js
2 Nov 00:55:14 - [nodemon] v1.4.1
2 Nov 00:55:14 - [nodemon] to restart at any time, enter `rs`
2 Nov 00:55:14 - [nodemon] watching: *.*
2 Nov 00:55:14 - [nodemon] starting `node helloWorld.js`
Server running at http://127.0.0.1:8000/
```

如上图所示，如果修改了 `helloWorld.js` 文件，nodemon 会重载应用。这可以避免由编辑、重载周期带来的开销，提升开发效率。

PM2——业界标准的任务执行器

虽然 forever 已经相当不错了，但是 PM2 比 forever 更胜一筹。通过 PM2，你可以管理

应用的整个生命周期，并且实现没有停机时间。只要通过简单的命令就可以伸缩应用。

PM2 也具备负载均衡的功能。

让我们一起看看以下例子：

```
var http = require('http');

var server = http.createServer(function (request, response) {
  console.log('called!');
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World\n");
});
server.listen(8000);
console.log("Server running at http://127.0.0.1:8000/");
```

这是一个相当简单的应用，我们通过 PM2 来运行它。

```
pm2 start helloWorld.js
```

通过以上命令，将得出与下图所示类似的输出：

```
[PM2] Starting helloWorld.js in fork_mode (1 instance)
[PM2] Done.
┌──────────┬────┬──────┬──────┬────────┬─────────┬────────┬──────────┐
│ App name │ id │ mode │ pid  │ status │ restart │ uptime │ memory   │ watching │
├──────────┼────┼──────┼──────┼────────┼─────────┼────────┼──────────┼──────────┤
│ helloWorld │ 0 │ fork │ 6858 │ online │ 0       │ 0s     │ 5.910 MB │ disabled │
└──────────┴────┴──────┴──────┴────────┴─────────┴────────┴──────────┴──────────┘
Use `pm2 show <id|name>` to get more details about an app
```

PM2 已经注册了一个名为 helloWorld 的应用。这个应用运行在 fork 模式下，该模式下 PM2 不进行负载均衡处理，而是简单地 fork 该应用，本例中该应用的 PID 为 6858。

现在输入命令 pm2 show 0，将得到 id 为 0 的应用的相关信息，如下图所示：

```
➜ ~ pm2 show 0
Describing process with id 0 - name helloWorld
┌──────────────────┬──────────────────────────────────────────────┐
│ status           │ online                                       │
│ name             │ helloWorld                                   │
│ id               │ 0                                            │
│ path             │ /Users/dgonzalez/helloWorld.js               │
│ args             │                                              │
│ exec cwd         │ /Users/dgonzalez                             │
│ error log path   │ /Users/dgonzalez/.pm2/logs/helloWorld-error-0.log │
│ out log path     │ /Users/dgonzalez/.pm2/logs/helloWorld-out-0.log │
│ pid path         │ /Users/dgonzalez/.pm2/pids/helloWorld-0.pid  │
│ mode             │ fork_mode                                    │
│ node v8 arguments │                                             │
│ watch & reload   │ ✘                                            │
│ interpreter      │ node                                         │
│ restarts         │ 0                                            │
│ unstable restarts │ 0                                           │
│ uptime           │ 2m                                           │
│ created at       │ 2015-11-02T01:23:45.434Z                     │
└──────────────────┴──────────────────────────────────────────────┘
Probes value
┌────────────┬─────────┐
│ Loop delay │ 0.94ms  │
└────────────┴─────────┘
```

通过这两个命令，我们已经可以相当全面地管理一个简单应用的执行情况。

从现在开始，PM2 能够确保你的应用一直处于运行状态，如果你的应用挂掉了，PM2 将对其进行重启。

我们也可以监控 PM2 下运行的应用列表：

```
pm2 monit
```

将得到如下输出：

```
◎ PM2 monitoring (To go further check out https://app.keymetrics.io)

• helloWorld                [                                   ] 0 %
[0] [fork_mode]             [|||||                              ] 27.340 MB
```

上图是 PM2 监控显示效果。在本例中，我们的系统中只有一个运行在 fork 模式下的应用，所以显得有些大材小用。

通过 `pm2 logs` 命令，可以查看输出日志，如下图所示：

```
PM2: 2015-11-02 01:14:38: App name:helloWorld id:3 disconnected
PM2: 2015-11-02 01:14:38: App name:helloWorld id:3 exited with code SIGTERM
PM2: 2015-11-02 01:14:38: Process with pid 5322 killed
PM2: 2015-11-02 01:14:44: Starting execution sequence in -cluster mode- for app name:helloWorld id:0
PM2: 2015-11-02 01:14:44: App name:helloWorld id:0 online
PM2: 2015-11-02 01:23:36: Stopping app:helloWorld id:0
PM2: 2015-11-02 01:23:36: App name:helloWorld id:0 disconnected
PM2: 2015-11-02 01:23:36: App name:helloWorld id:0 exited with code SIGTERM
PM2: 2015-11-02 01:23:37: Process with pid 6804 killed
PM2: 2015-11-02 01:23:45: Starting execution sequence in -fork mode- for app name:helloWorld id:0
PM2: 2015-11-02 01:23:45: App name:helloWorld id:0 online
PM2: 2015-11-02 01:31:33: Stopping app:helloWorld id:0
PM2: 2015-11-02 01:31:33: App name:helloWorld id:0 exited with code SIGINT
PM2: 2015-11-02 01:31:33: Process with pid 6858 killed
PM2: 2015-11-02 01:31:56: Starting execution sequence in -fork mode- for app name:helloWorld id:0
PM2: 2015-11-02 01:31:56: App name:helloWorld id:0 online

helloWorld-0 (out): Server running at http://127.0.0.1:8000/
helloWorld-0 (out): Server running at http://127.0.0.1:8000/
helloWorld-0 (out): Server running at http://127.0.0.1:8000/
helloWorld-0 (out): Server running at http://127.0.0.1:8000/
helloWorld-0 (out): Server running at http://127.0.0.1:8000/
helloWorld-0 (out): Server running at http://127.0.0.1:8000/
helloWorld-0 (out): Server running at http://127.0.0.1:8000/

[PM2] Streaming realtime logs for [all] processes



helloWorld-0 called!
helloWorld-0 called!
helloWorld-0 called!
helloWorld-0 called!
```

如你所见，PM2 相当出色。只需要很少的命令，就能够覆盖 90%的监控需求。然而，这还只是“冰山一角”。

PM2 还可以让你轻易地无缝重启应用：

```
pm2 reload all
```

这个命令可以确保你的应用能完成重启，并且无停机时间。PM2 会将你的请求放入队列，等待重启后的应用再次响应请求。还有一种更加细粒度的选择，可以通过加上应用名来指定需要重启的应用：

```
pm2 reload helloWorld
```

对于使用过多年 Apache、NGINX 以及 PHP-FPM 等 Web 服务器的人来说，这应该相当熟悉。

PM2 将启动一个控制进程和指定个数的工作进程（你的应用）。因此，单线程模型的 Node.js 也能享受到多核 CPU 带来的性能提升。

在切换到这个模式之前，我们得先停止之前的应用：

```
pm2 stop all
```

将得到如下所示的输出：

```
┌─────────────┬────┬──────┬─────┬─────────┬─────────┬────────┬────────┬──────────┐
│ App name    │ id │ mode │ pid │ status  │ restart │ uptime │ memory │ watching │
├─────────────┼────┼──────┼─────┼─────────┼─────────┼────────┼────────┼──────────┤
│ helloWorld  │ 0  │ fork │ 0   │ stopped │ 0       │ 0      │ 0 B    │ disabled │
└─────────────┴────┴──────┴─────┴─────────┴─────────┴────────┴────────┴──────────┘
Use `pm2 show <id|name>` to get more details about an app
```

PM2 会记住之前运行过的应用，因此，在重新使用集群模式运行应用之前，应该先让 PM2 抹去关于之前应用的记忆：

```
pm2 delete all
```

```
[PM2] Deleting all process
[PM2] deleteProcessId process id 0
┌─────────────┬────┬──────┬─────┬─────────┬─────────┬────────┬────────┬──────────┐
│ App name    │ id │ mode │ pid │ status  │ restart │ uptime │ memory │ watching │
└─────────────┴────┴──────┴─────┴─────────┴─────────┴────────┴────────┴──────────┘
Use `pm2 show <id|name>` to get more details about an app
```

现在，我们使用集群模式启动应用：

```
pm2 start helloWorld.js -i 3
```

```
[PM2] Starting helloWorld.js in cluster_mode (3 instances)
[PM2] Done.
┌─────────────┬────┬─────────┬──────┬────────┬─────────┬────────┬────────────┬──────────┐
│ App name    │ id │ mode    │ pid  │ status │ restart │ uptime │ memory     │ watching │
├─────────────┼────┼─────────┼──────┼────────┼─────────┼────────┼────────────┼──────────┤
│ helloWorld  │ 0  │ cluster │ 7477 │ online │ 0       │ 0s     │ 26.023 MB  │ disabled │
│ helloWorld  │ 1  │ cluster │ 7478 │ online │ 0       │ 0s     │ 26.316 MB  │ disabled │
│ helloWorld  │ 2  │ cluster │ 7479 │ online │ 0       │ 0s     │ 24.203 MB  │ disabled │
└─────────────┴────┴─────────┴──────┴────────┴─────────┴────────┴────────────┴──────────┘
Use `pm2 show <id|name>` to get more details about an app
```

PM2 充当控制主进程与 3 个工作进程之间的轮询调度器，因此它们可以并行地处理 3 个请求。我们能够自由地增加、减少工作进程：

```
pm2 scale helloWorld 2
```

通过执行上述命令，工作进程由 3 个减少为 2 个。

```
◇ PM2 monitoring (To go further check out https://app.keymetrics.io)

 • helloWorld                    [                    ] 0 %
[1] [cluster_mode]               [                    ] 30.133 MB

 • helloWorld                    [                    ] 0 %
[2] [cluster_mode]               [                    ] 30.477 MB
```

如你所见，我们能够毫不费力地配置应用，以做好生产准备。

现在，我们可以保存 PM2 的当前状态。这样一来，当重启服务器时，PM2 将会始终以守护进程的形式运行，我们的应用可以自动重新启动。

PM2 开放了编程接口，我们可以编写 Node.js 程序来管理之前例子中的所有手动过程。同时，可以通过读取 JSON 文件的方式来配置应用服务，这将在第 6 章中深入进行介绍，在第 6 章中我们将学习如何使用 PM2 和 Docker 部署 Node.js 应用。

小结

在本章中，你掌握了 Seneca 和 PM2 的基础知识。在第 4 章中，你可以使用它们搭建一个面向微服务的系统。

我们也证明了第 1 章中提及的概念对于解决实际问题是行之有效的，同时它们也能简化我们的工作。

在下一章中，我们将讨论如何拆解单块应用，本章中阐述的一些概念是学习下一章的必备前导知识。

3

从单块软件到微服务

在我的职业生涯中，曾服务过许多公司，其中绝大多数是金融服务公司，而这些我曾任职过的公司都会经历如下发展模式：

1. 一个公司由几个精通专项领域知识的人才组成，例如：保险、支付、信用卡等领域。
2. 随着公司的成长，不断出现新的需要快速响应的业务需求（例如，规范管理制度、解答大量小白客户的问题等），公司会因此进入一个自然增长阶段。
3. 接着，公司将会经历下一个增长阶段，在这个阶段，业务交易定义明晰，此时难以维护的单块软件已不能很好地完成对业务的建模。
4. 由于最初软件构建方式的限制，导致公司不断增加人力来解决该问题，伴随而来的是增长的痛苦与低下的效率。

本章不仅介绍如何避免上述情况的出现（失控的自然增长），还将介绍如何使用微服务来构建新系统。同时，本章也是本书的灵魂所在，我将会花一点篇幅来对亲身经历的工作经验进行提炼，并提出几点在第 4 章中将要遵循的原则。在第 4 章中，我们将会使用前面几个章节所学到的知识来构建一个完整的系统。

首先，我们拥有一个单块软件

有很大比重的（我估计在 90%左右）现代企业软件是按照单块软件的方式构建的。

运行于单一容器且开发周期严格定义的大型软件组件是完全违背敏捷开发原则的。敏捷开发的原则是：及早交付和频繁交付（`https://en.wikipedia.org/wiki/Release_early`, `_release_often`），如下所述。

- **及早交付**：你失败得越早，就越容易修复。如果你的软件开发周期持续了两年后才发布，将会有极大的风险与原始需求产生偏差，因为在这个过程中，需求可能会由于不合理而常常发生变更。
- **频繁交付**：频繁交付可以使软件干系人（stakeholders）了解开发进程，并且及时了解项目中的变更。从而能在几天内就修复错误，并且轻松发现系统优化点。

许多公司之所以使用大型软件组件而不是小块服务的结合，是因为这样做相当自然，其过程如下所示：

1. 开发者接到一个新需求。
2. 他在服务层已有的类中添加了一个新方法。
3. 写好的方法会暴露成 API，可以通过 HTTP、SOAP 或者其他协议访问该 API。

将这个过程乘以公司的开发者数量，得到的产物便叫作自然增长。自然增长指的是，由于缺乏充分的长期规划，在业务压力下，软件系统无计划、无控制地增长。这样的情况相当糟糕。

如何控制自然增长

控制自然增长的首要任务就是确保公司中的 IT 部门能与业务部门相匹配。通常大型公司并不将 IT 部门视为业务核心部门。

甚至，有些企业将他们的 IT 系统外包给其他公司来完成，这样他们可以将开发成本把控在一个确定的范围内，且并不重视 IT 系统的质量。因此，外包公司在构建软件系统的时候，只关注一件事：按照委托方指定的需求按时交付，哪怕需求是错的。

最终这将导致这些企业没有理想的生态系统来有计划地响应业务需求，只停留在逐个问题、逐个处理的原始阶段。这主要归咎于 IT 部门的管理者对于系统的构建几乎一无所知，并且常常忽视软件开发过程中的复杂性。

幸运的是，由于 IT 系统已成为世界上 99%的业务的驱动者，所以上述情况将有改善的趋势。但是，我们应该用明智的方法来构建 IT 系统。

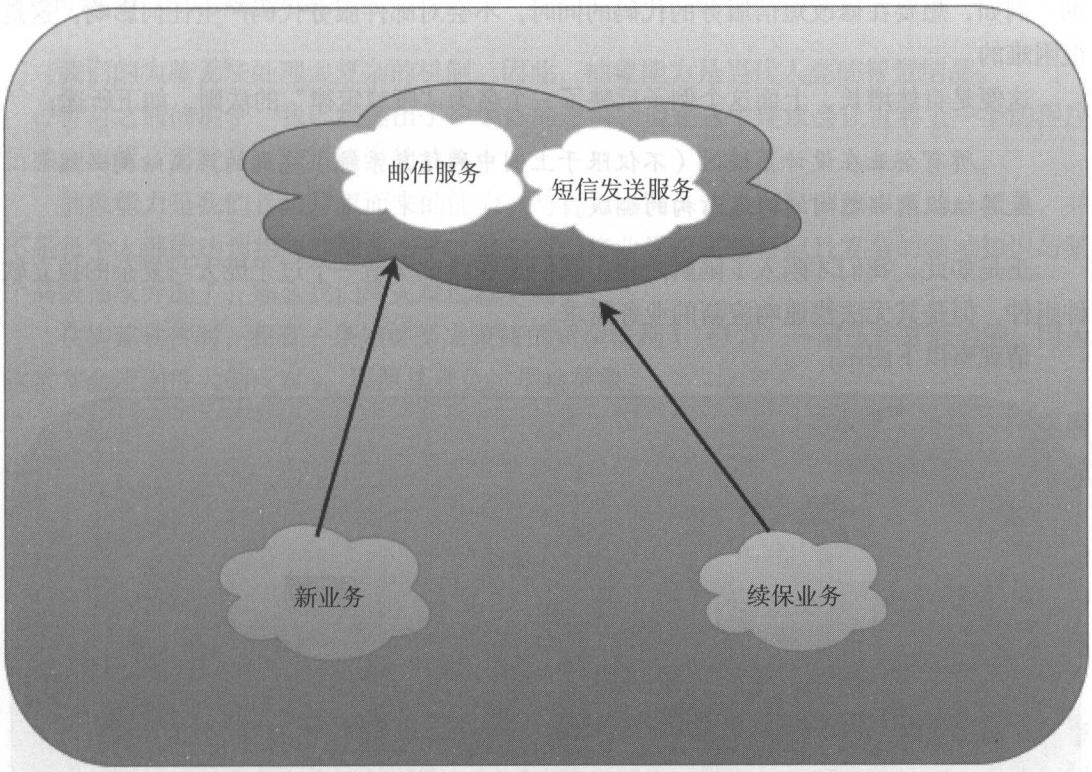
控制自然增长的第一要素就是让 IT 与业务干系人协同工作：培养非技术干系人成为成功的关键。

如果重新审视前面提到的这种少见的大型发布计划，我们是否能提出更好的解决方案呢？

答案是肯定的。可以将工作内容划分成若干个可控的软件工件，并为每个工件对应一个定义好的业务活动，并为它分配一个实体（entity）。

在这一阶段，还不需要将其暴露成微服务，但是，将逻辑保持在一个独立的、定义良好的、易于测试且解耦的模块中，将对我们未来改造应用带来极大的好处。

请考虑以下例子：



在一个保险系统中，某些用户的事项会显得比较紧急。虽然短信服务和邮件服务都是一种通信渠道，但是本质却不一样，因此，你或许会希望以不同的方式来运用它们。

调用服务分为以下两组高级实体。

- **新业务：**新客户在他们完成注册时会收到邮件。
- **续保业务：**老客户在保单需要续保时会收到短信提醒。

在某些情况下，系统不仅需要发送短信、邮件，还需要支持所有第三方通信服务。

乍看之下，这貌似是一个好主意。不管是短信服务还是邮件服务，最终都是一种通信

渠道，而且通信机制几乎 90%都是相同的，因此我们可以大量复用已有功能。

试想一下，如果我们突然想要把一个处理所有线下投递的第三方服务集成到现有系统中，会发生什么？

如果我们想要每周为客户推送那些他们感兴趣的新闻时事，又会发生什么？

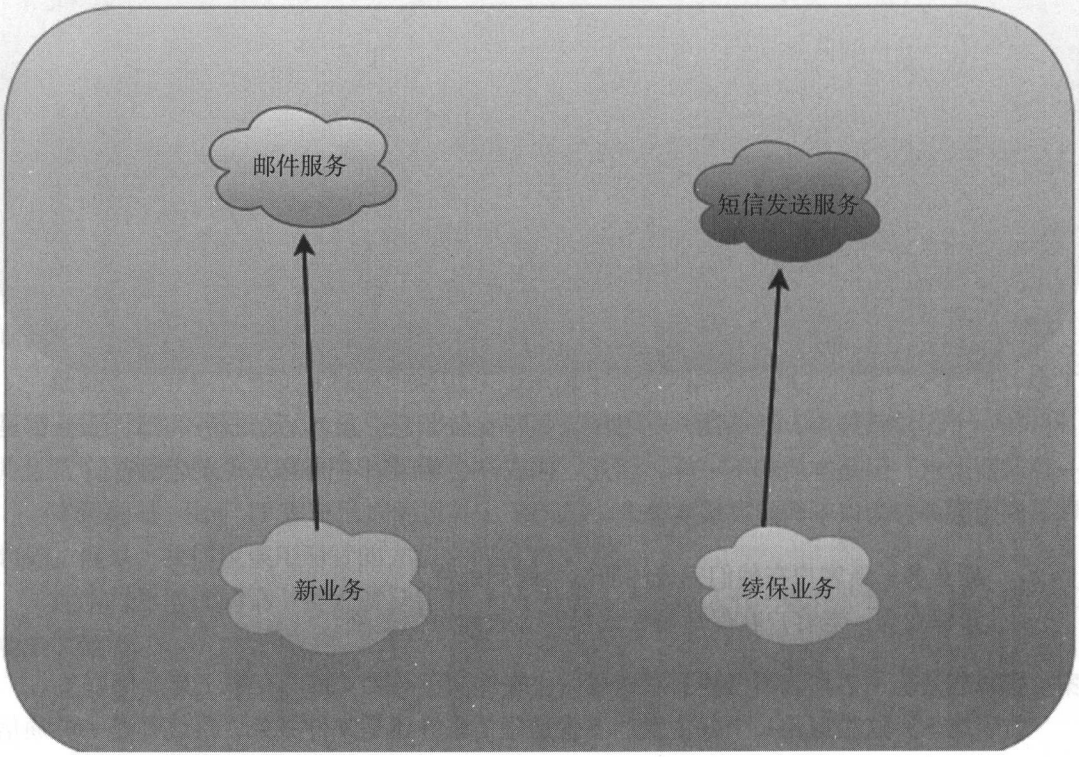
我们的服务系统将在失控的情况下增长，这让测试、发布与确保隔离性都变得更加困难。例如，想要在修改短信服务的代码的同时，不会对邮件服务代码产生任何影响，这是很困难的。

这便是自然增长，上面这个例子反映了一个名为“康威定律”的法则，如下所述：

> 所有企业在设计系统时（不仅限于上例中的信息系统）得到的产品结构必然是其组织内部之间的沟通结构的翻版。

正是如此，我们才陷入了陷阱之中。我们正在尝试塑造一个过于庞大与复杂的独立软件组件，但是其无法快速响应新的业务需求。

请观察以下图示：



现在，我们将每个通信渠道都封装在各自的服务体内（每个服务体随后将作为一个微服务进行部署），对于后续添加的新通信渠道我们也将做同样的处理。

这是解决自然增长的第一步：创建具有良好边界和具有单一职能的细粒度服务，其中单一职能表示将事情做得小而精。

多抽象才是过度抽象

我们的大脑无法处理太复杂的机制。因此，抽象能力是当代人类的智慧结晶。

针对之前的例子，我已经提出了一个好的建议，但是这个建议会让世界上一半的程序员感到困扰，那就是去除系统中的抽象。

抽象能力是我们日积月累而来的能力，与智力不同，它是可以训练出来的。但是，并不是每个人都能达到相同的抽象水平。如果我们将业界所需的特定且复杂的领域知识与某个高级抽象弄混了，那么我们离灾难也就不远了。

在构建软件时，我有一条始终努力秉持的黄金法则（“努力”一词是非常恰当的，因为这常常会遭到极大的反对），那便是避免过早地抽象。

回忆一下，你曾经多少次被一些简单的需求逼到了角落：假设需求是为解决 X 问题而编写一个程序，然而，你的团队成员已经对该问题进行了处理，并提前考虑了 X 的所有变体情况，做出了解决方案，但是并没有考虑这样做是否合理。接着，软件投入生产使用，某位干系人提出了一个你们从未考虑到的 X 的变体情况（例如之前的需求甚至是不正确的）。而现在，如果想要让这一个变体情况可解，得耗费你许多天的时间来进行大量的重构工作。

避免这个问题的方式很简单：*在没有重复使用 3 次以上的场景下避免使用抽象。*

以前面提到的通信系统为例，不要过多地考虑那些不可能出现的通信渠道，不然你将不得不对当前场景进行不必要的抽象。当拥有两个通信渠道时，你才需要开始考虑如何把这两个软件组件设计得更好，而当第三种通信渠道出现时，你就需要重构它们了。

记住，在构建微服务的时候，应该把每个微服务构建得足够小，这样可以在单个 sprint（大约两周时间）内进行重构。在如此短的周期内完成软件原型的好处在于，一旦有更明确的需求时，对原型进行重构是值得的，因为让相关干系人直接看到产品是最快敲定需求的方式。

Seneca 十分符合微服务的设计哲学，通过模式匹配，能够在对已有代码无影响的情况下扩展微服务已有的 API：我们的服务对扩展开放，对修改关闭（SOLID 原则之一），增加功能的同时不影响已有功能。在第 4 章中，我们将看到更完备的例子。

微服务的出现

微服务已被广泛接受。现在，许多公司对软件质量的重视程度提高。正如之前提到的，及早交付和频繁交付已成为软件开发成功的关键。

微服务通过模块化与专职化的方式帮助我们尽快地满足业务需求。每个小模块能够轻易地在几天内完成版本化与升级，同时它们也非常易于测试，因为它们拥有清晰、单一的目标（专职化），并且在编写它们时也保持了与系统其他部分的独立性（模块化）。

不幸的是，上述情况并不普遍。通常，大型软件系统在构建的时候，难以进行专职化与模块化。常规的做法是，构建单个大型软件组件来处理所有情况。而模块化使用得也非常少，因此我们需要从基础开始构建。

让我们通过代码进行讲解，如下所示：

```
module.exports = function(options) {

  var init = {}

  /**
   * 发送短信
   */
  init.sendSMS = function(destination, content) {
    // 发送短信代码实现
  }
  /**
   * 读取未读短信列表
   */
  init.readPendingSMS = function() {
    // 接收短信代码实现
    return listOfSms;
  }
  /**
   * 发送邮件
   */
  init.sendEmail = function(subject, content) {
    // 发送邮件代码实现
  }
```

```
    /**
     * 读取未读邮件列表
     */
    init.readPendingEmails = function() {
      // 读取未读邮件代码实现
      return listOfEmails;
    }

    /**
     * 这段代码将已读邮件打上标记，这样它们就不会被
     * readPendingEmails函数重复读取。
     */
    init.markEmailAsRead = function(messageId) {
      // 将信息标记为已读代码实现
    }

    /**
     * 这个函数将待印刷及邮寄的文件放入队列中
     */
    init.queuePost = function(document) {
      // 邮寄队列代码的实现
    }

    return init;
}
```

如你所见，该模块可以被简单地称为通信服务，并且用户很容易猜到它具有哪些功能。它管理着邮件、短信以及邮寄等通信方式。

通信渠道或许已经足够多了。如果开发者继续在这个服务中加入其他通信渠道的相关方法，那么该服务的增长就进入了失控阶段。这就是单块软件的关键问题所在：限界上下文跨越了多个领域，从而影响到软件的功能性和可维护性这两方面的质量。

如果你是一名软件开发者，那么你会发现一个很明显的问题：这个模块的内聚性太差了。

虽然该模块可以运作一段时日，但是我们必须从现在开始就改变思维模式，应该构建微小的、可伸缩的、自治的独立模块。当一个模块处理太多事情，诸如邮件、短信和邮寄时，它的内聚性就会变得很差。

如果我们要加入其他通信渠道，例如 Twitter 和 Facebook 的通知，那又会发生什么？

服务会进入失控增长阶段。你将拥有一个难以重构、测试和修改的大型模块，而不是

多个小型功能性组件。让我们回顾一下第 2 章中提到的 SOLID 设计原则。

- **单一职责原则**：单个模块处理了太多任务。
- **开放封闭原则（对扩展开放，对修改关闭）**：单块模块在增加新功能时需要修改模块代码，这可能会改变公用代码。
- **里氏替换原则**：这里我们仍然跳过该原则。
- **接口分离原则**：单块模块中并没有定义接口，只有一系列功能的堆砌实现。
- **依赖注入原则**：并没有使用依赖注入。模块需要显式通过调用代码来构建。

如果不进行测试的话，会出现很多难以预料的结果。因此，我们通过 Seneca 将它切分成数个小模块。首先，邮件模块（email.js）如下所示：

```
module.exports = function (options) {

  /**
   * 发送邮件
   */
  this.add({channel: 'email', action: 'send'}, function(msg,
    respond) {
    // 发送邮件代码实现
    respond(null, {...});
  });

  /**
   * 获取未读邮件列表
   */
  this.add({channel: 'email', action: 'pending'}, function(msg,
    respond) {
    // 读取未读邮件代码实现
    respond(null, {...});
  });

  /**
   * 将信息标记为已读
   */
  this.add({channel: 'email', action: 'read'}, function(msg,
    respond) {
    // 标记信息为已读代码实现
```

```
    respond(null, {...});
  });
}
```

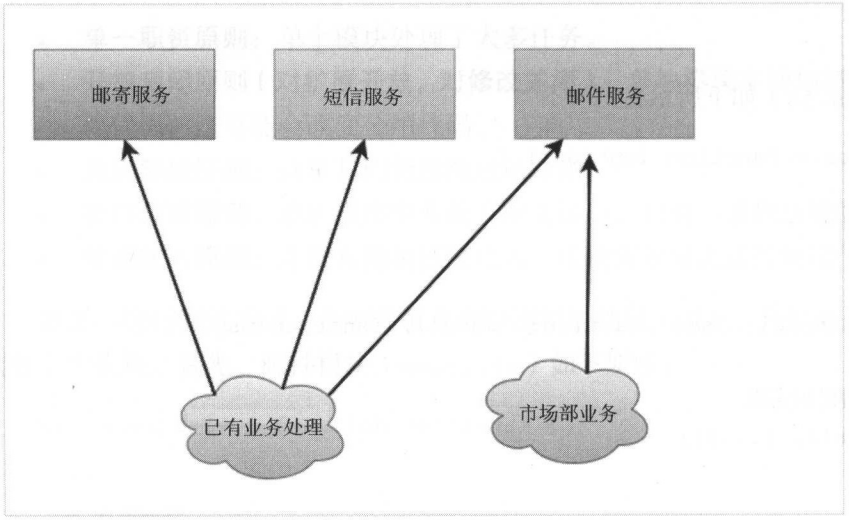
短信模块（sms.js）如下所示：

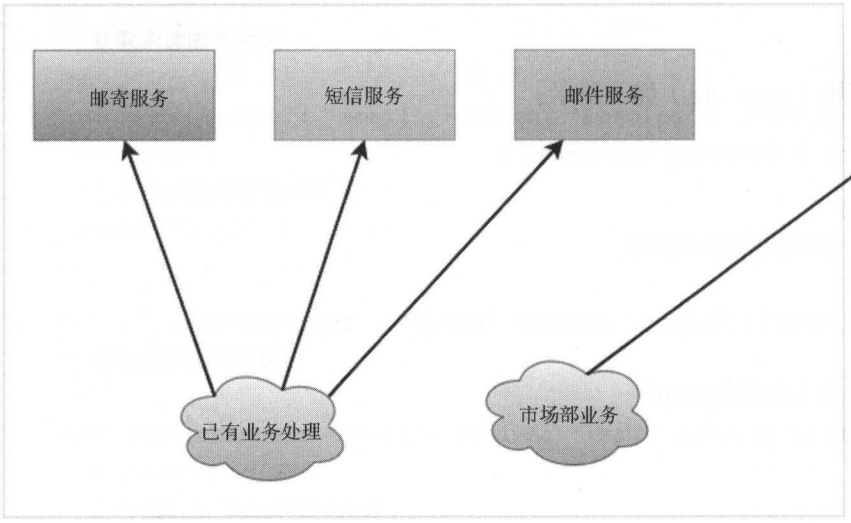
```
module.exports = function (options) {

  /**
   * 发送短信
   */
  this.add({channel: 'sms', action: 'send'}, function(msg,
    respond) {
    // 发送短信代码实现
    respond(null, {...});
  });

  /**
   * 获取未读短信
   */
  this.add({channel: 'sms', action: 'pending'}, function(msg,
    respond) {
    // 读取未读短信代码实现
    respond(null, {...});
  });
}
```

最后，邮寄模块（post.js）如下所示：

```
module.exports = function (options) {

  /**
   * 将待打印、邮寄的信息放入队列
   */
  this.add({channel: 'post', action: 'queue'}, function(msg,
    respond) {
    // 邮寄信息放入队列代码实现
    respond(null, {...});
  });
}
```

以下图示展示了新的模块结构：





现在，我们有了三个模块。这三个模块各自处理指定业务，并且不会干涉其他模块。因此，我们构建了三个高内聚的模块。

让我们使用以下方式运行上述代码：

```
var seneca = require("seneca")()
        .use("email")
        .use("sms")
        .use("post");

seneca.listen({port: 1932, host: "10.0.0.7"});
```

为了简单起见，我们将服务绑定在 IP：10.0.0.7 上，通过 1932 端口监听请求。如你所见，我们并没有指定任何配置文件，只要通过模块的名字来指定使用它，Seneca 将完成余下的工作。

启动 Seneca，确认它是否加载了我们定义的这三个插件：

```
node index.js --seneca.log.all | grep DEFINE
```

该命令将会得到与下图类似的输出结果：

```
2015-11-12T22:57:24.720Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin basic      DEFINE {}
2015-11-12T22:57:24.790Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin transport DEFINE {}
2015-11-12T22:57:24.832Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin web       DEFINE {}
2015-11-12T22:57:24.847Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin mem-store DEFINE {}
2015-11-12T22:57:24.860Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin email      DEFINE {}
2015-11-12T22:57:24.872Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin sms        DEFINE {}
2015-11-12T22:57:24.882Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin post       DEFINE {}
```

如果你还记得，第 2 章中曾提到过 Seneca 会默认加载一些插件，比如：basic、transport、web 以及 mem-store。默认加载的插件只需简单配置即可在 Seneca 中使用。我们将在第 4 章中看到对它们的使用以及哪些配置是必要的。比如 mem-store 加载之后，程序执行期间只会将数据保存在内存中，不会对其进行持久化存储。除了以上标准插件外，我们还可以看见 Seneca 额外加载了三个我们创建的插件：email、sms 和 post。

如你所见，只要知道了框架是如何工作的，那么使用 Seneca 编写的服务程序就相当好理解。在这个例子中，我以插件的形式来编写代码，这样得到的代码可以在不同机器上的不同 Seneca 实例中运行。这得益于 Seneca 提供了透明的传输机制，允许我们可以像对待微服务一样对单块应用中的各个部分进行快速重新部署和扩展，如下所述：

- 新版本易于测试，因为对于邮件功能的变更只会影响发送邮件的功能。

- 新版本易于伸缩。我们将在下一章中看到，复制一个服务和“配置一个新服务器并将我们的 Seneca 客户端指向它”一样简单。
- 同样的，新版本也易于维护，因为单功能软件相当容易理解和修改。

微服务的缺陷

通过微服务，我们解决了现代企业中面临的最大问题，但这并不意味着不再有其他问题。微服务经常会引发一些我们不容易预见的问题。

微服务的首要问题是，人工操作的开销会使微服务带来的好处大打折扣。当你在设计一个系统的时候，应该总是考虑一个问题：如何自动化处理。自动化是解决这一问题的关键。

微服务的第二个缺点是应用的不一致性。例如，一个团队的理想实践方式被另一个团队弃如敝履（尤其是异常处理上）。这样，会在不同团队间产生无形的隔阂，不利于工程师之间的交流。

最后但并非不重要的一点是，微服务引入了更多的通信复杂性，这样可能会导致安全问题。相比原来我们只需控制单个应用服务及其与外界的通信，现在面临着多个需要彼此通信的服务器。

分割单块软件

考虑以下例子，贵公司的市场部想组织一次激进的邮件宣传活动，因此带来的邮件处理峰值将影响到日常的邮件发送功能。在高访问压力下，邮件发送将会出现延迟，这将会给我们造成其他问题。

幸运的是，我们使用了之前提到的方式来构建服务系统。小型的 Seneca 模块都是一个个高内聚、低耦合的插件。

这样一来，针对以上问题的解决方案变得相当简单，即只需在多台机器上部署邮件服务（`email.js`）：

```
var seneca = require("seneca")().use("email");
seneca.listen({port: 1932, host: "new-email-service-ip"});
```

同时，创建一个指向服务端的 Seneca 客户端：

```
var seneca = require("seneca")()
    .use("email")
```

```
    .use("sms")
    .use("post");
seneca.listen({port: 1932, host: "10.0.0.7"});

// 通过“seneca”与已有的邮件服务交互

var senecaEmail = require("seneca").client({host: "new-email-
  service-ip", port: 1932});

// 通过“senecaEmail”与新的邮件服务交互
```

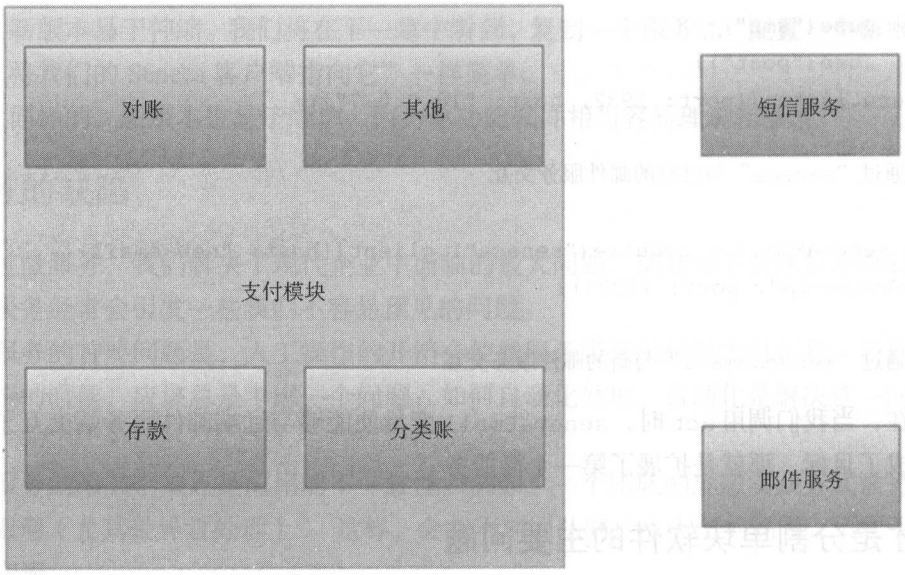
现在，当我们调用 act 时，senecaEmail 变量便能够与远端邮件服务端交互了，我们也就达成了目标，那就是扩展了第一个微服务。

数据才是分割单块软件的主要问题

数据存储存在问题，如果你的应用已经自然增长了多年，那么数据库也是如此，这将导致难以处理数据库的重大变更。

微服务需要维护它们自己的数据。保持服务数据的局部性是保证系统升级时具有灵活性的关键，但是这并不总是能够实现。例如，在构建金融服务系统时，面向微服务架构的主要痛点就是缺乏事务性。对于跟钱打交道的组件来说，必须保证每一次独立操作的数据一致性而不是最终一致性。如果一个客户在一家金融公司存了一笔钱，那么软件中的账户余额必须与银行里的钱数保持一致，否则就会对账失败。不仅如此，如果你的公司是一个受监管的实体，这将严重影响业务的连续性。

使用微服务搭建金融系统的主要原则是：涉及金钱处理的部分不要过于微服务化，而对于其他例如邮件、短信和用户注册等辅助模块，则适合微服务化。如下图所示：



在上图中可以看到，支付模块是一个大型的微服务，我们并没有将它拆解成更小型的服务。这只会对运维造成影响，对之前提到的模块化并不会有任何阻碍。从 ATM 机中提款的操作必须作为一个原子操作（要么成功、要么失败，没有中间状态），这种原子处理并不会影响应用的代码组织形式，我们仍然可以对服务进行模块化，只是该模块的事务范围会横跨多个服务。

组织架构适配

如果一个公司采用微服务来构建软件系统，那么每个干系人都需要参与决策。

微服务是一次重大的范式转换。通常，大型组织倾向于使用相当传统的方式来构建软件系统。每个重大发布需要经历数月的研发周期，之后需要一个完备的质量保证阶段以及数小时的部署阶段。

当一个公司选择使用面向微服务的架构时，方法论就会发生完全的改变：每个小团队负责各自的小功能点，包括它们的构建、测试和部署。每个团队各司其职，并且能够处理好各自负责的单一事项（一个微服务，或更确切地说是数个微服务），每个团队成员将熟练掌握构建软件系统的相关技术与领域知识。

这通常被称为跨职能团队。这是一个由少数人组成的工作单元，他们都具备了构建高质量软件组件的能力。

有一点值得注意的是，团队成员应当掌握必要的领域知识来理解业务需求。

在我的职业生涯中，大多数导致公司失败的主要问题不外乎以下这点（就我个人观点而言）。首先，有一种观点认为开发者是“堆砖器”，即可以在没有提前沟通的情况下却依然能神奇地理解业务流。而且，还有观点认为，如果一个开发者一周可以完成 X 量级的工作，那么 10 个开发者一周就可以交付 $10X$ 量级的产量。这些观点都是错误的。

为了保持高效以及考虑到康威定律在改变业务流程方面对系统的影响，构建微服务的跨职能团队中的成员必须熟练掌握（不仅仅是了解）相关领域知识。

每当谈及微服务的组织架构适配时，自治才是关键因素。为了保证构建微服务的敏捷性，每个团队都必须保持自治，这也意味着要确保技术的自主选择权，如下所示：

- 使用的语言。
- 代码规范。
- 解决问题的模式。
- 各类工具的选择，比如软件的构建、测试、调试及部署工具。

这是非常重要的部分，因为这是我们需要定义公司如何构建软件，并且可能会引入工程问题的部分。

举个例子，我们来看看代码规范问题，如下所示：

- 我们需要在所有团队内都保持统一的代码规范吗？
- 我们应该让每个团队都有各自的代码规范吗？

一般来说，我偏好于“80%准则”：80%的完美度已经足以涵盖 100%的使用场景。这意味着放宽代码规范（也适用于其他领域），以及允许一定程度的不完美或者个性化，都有助于减少团队间的摩擦，也能让工程师能够尽快关注到那些极少数的重要准则，比如日志策略以及异常处理。

如果你的代码规范过于复杂，那么在某个团队想要向其他团队的微服务提交代码时就会遇到很多阻力（切记，一个团队拥有自己的服务，但是其他团队的成员也可以对这个服务做出贡献）。

小结

在本章中，我们讨论了如何构建可以根据业务需求切分为微服务的单块应用。正如你所学到的，为了构建出高质量的软件，我们必须牢记原子性、一致性、隔离性、持久性（ACID）

这 4 个设计原则。

你同时也了解到，我们不能假设总能从零开始去设计一个系统，因此，必须妥善地为现有的系统构建新模块并重构已有模块，以此满足业务需求的灵活性和弹性。

我们还简短地介绍了以实现单块软件为目的而设计的数据库。它们是单块软件切分为微服务时最大的痛点，因为，将整体数据切分到各个本地数据库通常需要将系统关闭数小时。就 NoSQL 数据库对整个数据存储格局的改变趋势这一话题，完全可以独立成册进行阐述。

最后，我们讨论了如何协调公司内各个工程师团队之间的合作，从而高效地保持系统的弹性和灵活性，以满足敏捷开发的需求。同时，我们还讨论了康威定律是如何对单块系统转换成面向微服务架构产生影响的。

在下一章中，我们将应用前三章中讨论的原则和大量常识，构建出一个基于微服务的完整的可用系统。

4

编写你的第一个 Node.js 微服务

我们之前已经学习了如何搭建健壮的基于微服务的软件，现在是时候将理论知识用于实践了。在本章中，我们将使用 Seneca 以及其他能让我们能得益于微服务特性的框架，来构建一个基于微服务的电子商务软件。

微电子商务概览

本章包括以下内容：

- 编写微服务
- 调整微服务规模
- 创建 API
- 整合 Seneca 与 Express
- 通过 Seneca 持久化数据

在本章中，我们将编写一个完整的（几乎是）、简化的、基于微服务的电子商务解决方案。所谓的完整表示概念上的完整，但是显而易见，从生产的角度上看它并不完整。因为要处理所有可能的业务流，得花费好几本书的篇幅。

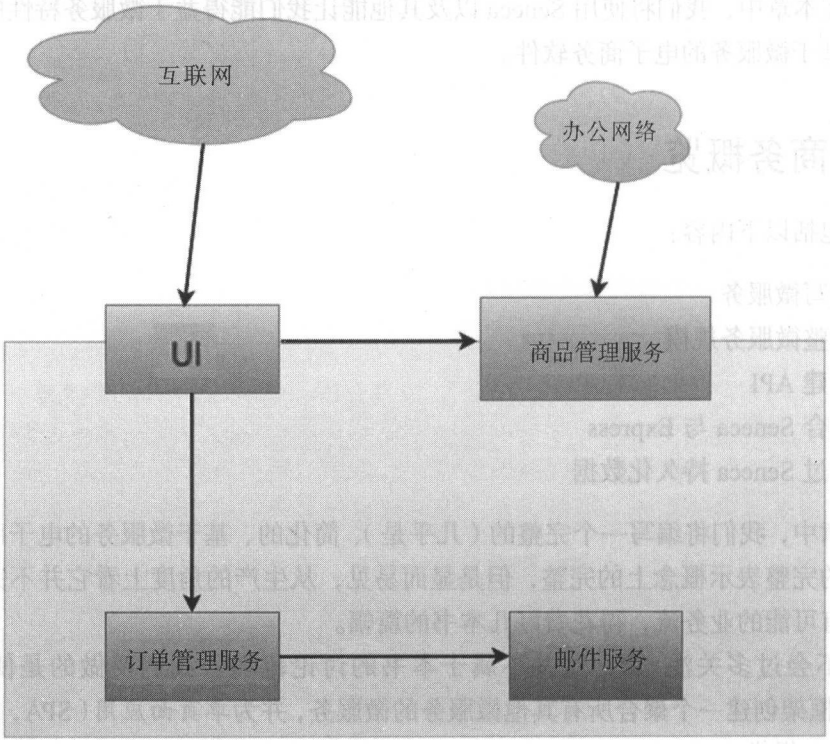
本章不会过多关注 UI，这并不属于本书的讨论范畴。我们要做的是使用流行的 JavaScript 框架创建一个聚合所有其他微服务的微服务，并为单页面应用（SPA，Single-Page Application）提供 API。

在本章中，我们将深入讨论以下 4 个微服务。

- **商品管理服务：**负责添加、编辑和删除数据库中的商品信息，以及给客户提供商品信息。本服务只会对部分用户开放具有添加和删除商品功能的管理页面。
- **订单管理服务：**负责管理订单与账单。
- **邮件服务：**负责向客户发送邮件。
- **UI：**负责给某个单页面应用提供其他微服务的特性，但是本章中我们只构建 JSON 接口。

我们将会用到之前学过的理论知识来搭建上述 4 个微服务。到本章结束，我们将能够识别出在搭建过程中最常见的陷阱。注意，本书的主旨不是让你成为微服务或者 Node.js 的专家，而是向你介绍需要你自学的微服务构建工具，让你理解最佳设计原则以及认识最常见的陷阱。

微服务部署图如下所示：



从上图中可以看出，我们公司（框内）将部分微服务对外界隐藏，分别对不同的网络暴露了不同服务：

- UI 将直接暴露在互联网上，所有人都可以访问它。
- **商品管理服务**管理电子商务系统中的产品信息。它有以下两个对外接口：
  - 为 UI 提供数据的 Seneca 服务端。
  - 公司内部用于创建、更新和删除商品信息的 JSON API。
- **邮件服务**是我们与客户的通信渠道。我们将通过该微服务阐述 Seneca 的优点，并会举例说明，当一个微服务宕机时，如何保证最终一致性和进行系统降级。
- **订单管理服务**负责处理客户的订单。通过这个微服务，我们将讨论如何进行微服务数据的本地化，而不再是以系统全局共享的方式管理数据。你无法直接在本机数据库中获取商品名字或价格，而是需要通过其他微服务获取。

如你所见，这个系统中并没有客户或员工信息管理功能，但是，基于上述的 4 个微服务，我们将能够深入微服务架构的核心理念。Seneca 拥有非常强大的数据存储与传输插件系统，使其能够很方便地与不同的数据存储系统、传输系统对接。

在本书中，我们将使用 MongoDB 作为所有微服务的数据存储层。Seneca 提供了开箱即用的内存数据库插件，使用者可以直接开始编码。但是，内存数据库只能暂存数据，并不能持久化存储。

商品管理服务——双重核心

商品管理服务是我们系统的核心。我知道你在想什么：微服务应该是足够小（微），并且是分布式的（无中心点）。但是，你还是应该设置一个概念上的中心。否则，最终将得到一个散乱的系统，并且有追溯性问题（我们将在之后进一步讨论这一问题）。

因为可以与 Express 直接整合，在 Seneca 上构建双重 API 是相当容易的。通过 Express，UI 可以向外界提供以下功能：编辑、添加、删除商品等，它是一个非常方便的框架，易于学习，并且可以很好地与 Seneca 整合。Express 实际上还是基于 Node.js 的网络应用的标准框架，因此，遇到问题时寻求解决方案也相当容易。

同时，商品管理服务还可以通过 Seneca TCP（Seneca 默认加载插件）让私有部分对外提供服务，因此内部网络中其他微服务（尤其是 UI）就能够访问到商品目录列表。

商品管理服务将是小而具有内聚性的（它只管理商品），并且具备可伸缩性。虽然微小，但是该服务拥有处理电子商务系统中商品信息的所有能力。

我们需要做的第一件事情是定义商品管理服务的功能，如下所示：

- 获取数据库中所有商品信息。这在实际生产环境中或许不是一个好方法（因为一般在生产系统中可能会需要分页），但是在我们的例子中是可行的。
- 根据指定分类获取商品信息列表。与前一个功能类似，在生产系统中需要对结果进行分页。
- 根据指定 ID 获取商品信息。
- 将商品信息添加到数据库（本例中使用 MongoDB 作为数据库）。该功能会使用到 Seneca 的数据抽象能力，将微服务与实际存储系统解耦：我们能够（在理论上）轻易地将底层数据库从 Mongo 切换到其他存储系统。
- 删除商品信息。同样使用到 Seneca 的数据抽象。
- 编辑商品信息。

我们的商品信息包含以下字段：名字、分类、描述以及价格。如你所见，这有一点过于简单，但是足以帮助我们理解复杂的微服务世界。

我们的商品管理微服务采用 MongoDB（`https://www.mongodb.org/`）作为存储方案。MongoDB 是一种面向文档（document）、无模式的数据库，在数据存储上有很大的灵活性，例如商品信息最终以文档的形式存储。MongoDB 是 Node.js 的绝佳选择，它存储的对象为 JSON（JavaScript Object Notation）格式，这是一种为 JavaScript 创建的格式标准，所以和 Node.js 十分匹配。

如果想了解更多关于 MongoDB 的知识，可以从它的官网获取到许多有用信息。现在，让我们开始编写微服务的功能。

获取商品信息

为了获取商品信息，我们会直接从数据库中获取所有的商品信息返回给接口。在这种情况下，我们没有创建任何分页机制。但是，通常情况下，数据分页是避免数据库（也可能是应用，但主要是数据库）性能问题的好方法。

让我们来看看以下代码：

```
/**
 * 获取所有商品列表
```

```
 */
seneca.add({area: "product", action: "fetch"}, function(args,
  done) {
  var products = this.make("products");
  products.list$({}, done);
});
```

这样，我们在 Seneca 中添加了一个可以返回数据库中所有商品信息的模式（pattern）。函数 products.list$()有以下两个入参：

- 查询条件。
- 一个可以处理错误和结果对象的函数（记住错误优先回调法则）。

Seneca 使用 $ 标记关键函数，例如：list$、save$等。在给对象属性命名时，只要你的命名仅包含数字与字母的组合，就可以避免命名上的冲突。

通过 seneca.add()方法，我们将 done 函数传递给 list$方法。因为 Seneca 符合错误优先回调法则，所以这种方法有效。它相当于以下代码的简写：

```
seneca.add({area: "product", action: "fetch"}, function(args,
  done) {
  var products = this.make("products");
  products.list$({}, function(err, result) {
    done(err, result);
  });
});
```

获取指定类别的商品

获取指定类别的商品与获取所有商品信息非常类似。唯一的区别是，在获取指定类别商品时需要传入类别参数，根据参数来筛选结果。

让我们看看以下代码：

```
/**
 * 获取指定类别的商品列表
 */
seneca.add({area: "product", action: "fetch", criteria:
  "byCategory"}, function(args, done) {
  var products = this.make("products");
```

```
    products.list$({category: args.category}, done);
});
```

高级开发人员心里可能会有这么一个问题：这难道不是一个会受到注入攻击的典型场景吗？当然不是。Seneca 有足够的能力去防止注入攻击，因此我们并不用担心这个问题。

如你所见，与上一段代码相比，这段代码唯一明显的区别是多了一个 `category` 入参，该参数将会被委派给 Seneca 的数据抽象层，并且根据底层存储系统生成相应的查询语句。这种功能对于微服务来说非常强大。前几章中我们多次提到耦合问题，仿佛它是“万恶之源”，然而，Seneca 可以优雅地处理这个问题，它提供了一个需要被不同存储插件共同遵循的规约。在以上代码中，`list$`正是规约的一部分。如果你合理地使用 Seneca 的存储层，那么替换微服务的存储引擎（你是否曾经尝试将你的部分数据迁移到 MongoDB）只需要进行一些相关配置即可。

根据 ID 获取商品

根据 ID 获取商品信息是最必要的功能之一，同时它的实现也存在难点。所谓的难点，并不是在编码层面。其代码如下：

```
/**
 * 根据id获取商品
 */
seneca.add({area: "product", action: "fetch", criteria: "byId"},
  function(args, done) {
  var product = this.make("products");
  product.load$(args.id, done);
});
```

该功能实现的难点在于如何生成 `id`。`id` 的生成是应用与数据库的一个关联点。Mongo 通过哈希生成一个复合 ID，而 MySQL 通常采用一个自增的整型数字来唯一标识每一行记录。如果想要将应用的存储系统从 MongoDB 切换为 MySQL，首先需要解决的问题是如何将以下哈希值转换为一个整型数字：

```
e777d434a849760a1303b7f9f989e33a
```

在 99%的场景下，这都没什么问题。但是，必须小心，尤其是对 ID 的存储。因为前面的章节提到过，每个微服务的数据都应该存放在本地，这意味着改变某个实体的 ID 数据类型需要同时改变其他数据库中相关联的 ID 类型。

添加商品

添加商品的功能非常容易实现。只需要创建一个商品信息，将其存入数据库即可：

```
/**
 * 添加商品
 */
seneca.add({area: "product", action: "add"}, function(args, done) {
  var products = this.make("products");
  products.category = args.category;
  products.name = args.name;
  products.description = args.description;
  products.category = args.category;
  products.price = args.price
  products.save$(function(err, product) {
    done(err, products.data$(false));
  });
});
```

在上述代码中，我们使用了 Seneca 的辅助方法 `product.data$(false)`，通过它我们能够获取所需的实体数据，并且返回的数据中不会包含命名空间（域）、实体名、库名等我们并不关心的元数据信息。

删除商品

删除商品的操作通常根据 `id` 来完成：根据指定主键来定位和删除指定商品，代码如下：

```
/**
 * 根据id删除指定商品
 */
seneca.add({area: "product", action: "remove"}, function(args,
  done) {
  var product = this.make("products");
  product.remove$(args.id, function(err) {
  done(err, null);
  });
});
```

在这段代码中，除了错误发生时会返回错误信息之外，不会返回其他任何信息；因此，对于调用终端来说，只要没有错误信息返回就表明删除动作成功。

编辑商品

我们需要提供一个编辑商品的功能，代码如下：

```
/**
 * 通过id获取商品信息并编辑
 */
seneca.edit({area: "product", action: "edit"}, function(args,
  done) {
  seneca.act({area: "product", action: "fetch", criteria: "byId",
  id: args.id}, function(err, result) {
  result.data$(
  {
    name: args.name,
    category: args.category,
    description: args.description,
    price: args.price
  }
  );
  result.save$(function(err, product){
    done(product.data$(false));
    });
  });
});
```

这是一个值得注意的场景：在编辑商品之前，需要先根据 id 取出商品信息，这个方法我们已经实现了。因此，需要做的是，取出商品信息后，将传入的数据放入相应字段并且保存。

这是 Seneca 引入代码复用的好方法：你可以在一个 action 中代理另一个 action 的调用，并且在包装 action 中处理结果。

整合各模块

正如我们之前提到的，商品管理服务具有双重性：一面是使用 Seneca 通过 TCP 传输

数据给其他微服务，另一面是通过 Express（Node.js 的 Web 应用框架）以 REST 风格提供服务。我们将所有代码整合如下：

```
var plugin = function(options) {
  var seneca = this;

  /**
   * 获取所有商品列表
   */
  seneca.add({area: "product", action: "fetch"}, function(args,
    done) {
    var products = this.make("products");
    products.list$({}, done);
  });

  /**
   * 根据分类获取商品列表
   */
  seneca.add({area: "product", action: "fetch", criteria:
    "byCategory"}, function(args, done) {
    var products = this.make("products");
    products.list$({category: args.category}, done);
  });

  /**
   * 根据id获取商品
   */
  seneca.add({area: "product", action: "fetch", criteria: "byId"},
    function(args, done) {
    var product = this.make("products");
    product.load$(args.id, done);
  });

  /**
   * 添加商品
   */
  seneca.add({area: "product", action: "add"}, function(args,
```

```
  done) {
  var products = this.make("products");
  products.category = args.category;
  products.name = args.name;
  products.description = args.description;
  products.category = args.category;
  products.price = args.price
  products.save$(function(err, product) {
    done(err, products.data$(false));
  });
});

/**
 * 根据id删除商品
 */
seneca.add({area: "product", action: "remove"}, function(args,
  done) {
  var product = this.make("products");
  product.remove$(args.id, function(err) {
    done(err, null);
  });
});

/**
 * 根据id获取商品信息并编辑
 */
seneca.add({area: "product", action: "edit"}, function(args,
  done) {
  seneca.act({area: "product", action: "fetch", criteria:
    "byId", id: args.id}, function(err, result) {
    result.data$(
      {
        name: args.name,
        category: args.category,
        description: args.description,
        price: args.price
      }
```

```
      );
      result.save$(function(err, product){
        done(err, product.data$(false));
      });
    });
  });
}
module.exports = plugin;

var seneca = require("seneca")();
seneca.use(plugin);
seneca.use("mongo-store", {
  name: "seneca",
  host: "127.0.0.1",
  port: "27017"
});

seneca.ready(function(err){

  seneca.act('role:web',{use:{
    prefix: '/products',
    pin: {area:'product',action:'*'},
    map:{
    fetch: {GET:true},
    edit: {GET:false,POST:true},
    delete: {GET: false, DELETE: true}
    }
  }});
  var express = require('express');
  var app = express();
  app.use(require("body-parser").json());

  // Seneca与Express集成
  app.use( seneca.export('web') );

  app.listen(3000);

});
```

现在，我们来分析上述代码。

我们构建了一个 Seneca 插件，该插件能够被不同的微服务复用，并且包含了前文中提到的商品管理服务需要的所有方法定义。

以上代码可以分为下列两个部分：

- 首先，和 Mongo 建立连接，指定 Mongo 为本地数据库。我们使用“mongo-store”插件实现以上功能（插件代码地址为 `https://github.com/senecajs/seneca-mongo-store`[译注1]，作者是 Richard Rodger，同时也是 Seneca 的作者）。
- 第二部分对于我们来说是个新内容。如果你曾经使用过 jQuery，可能会产生熟悉感。回调函数 `seneca.ready()` 需要处理调用 API 前 Seneca 和 Mongo 可能没有建立连接的情况。同时，它也包含了集成 Express 和 Seneca 的代码。

以下是应用的 `package.json` 文件的配置内容：

```
{
  "name": "Product Manager",
  "version": "1.0.0",
  "description": "Product Management sub-system",
  "main": "index.js",
  "keywords": [
    "microservices",
    "products"
  ],
  "author": "David Gonzalez",
  "license": "ISC",
  "dependencies": {
  "body-parser": "^1.14.1",
  "debug": "^2.2.0",
  "express": "^4.13.3",
  "seneca": "^0.8.0",
  "seneca-mongo-store": "^0.2.0",
  "type-is": "^1.6.10"
  }
}
```

译注1 原书地址是 https://github.com/rjrodger/seneca-mongo-store，目前已无法访问。

上述内容引入了微服务运行时需要的库和相关配置。

集成 Express 与 Seneca——如何创建 REST API

Seneca 与 Express 的集成相当容易，代码如下：

```
seneca.act('role:web',{use:{
  prefix: '/products',
  pin: {area:'product',action:'*'},
  map:{
  fetch: {GET:true},
  edit: {PUT:true},
  delete: {GET: false, DELETE: true}
  }
}});
var express = require('express');
var app = express();
app.use(require("body-parser").json());

// 这一步对Seneca与Express进行了集成
app.use( seneca.export('web') );

app.listen(3000);
```

这段前文出现过的代码，提供了下列三个 REST 服务端点：

```
/products/fetch
/products/edit
/products/delete
```

让我们对它们进行一下解释。

首先，我们指定 Seneca 根据配置执行 `role:web` action。配置信息中使用了`/product` 作为所有 URL 的前缀，同时将相关 action 与匹配模式`{area: "product", action: "*"}` 绑定。虽然我们在本书中第一次接触这种方式，但是这确实是 Seneca 中将 action 与 URL 绑定的一种可行方式，指明 `product` 这个 `area` 的处理程序。也就是说，`/products/fetch` 将与`{area: 'products',action: 'fetch'}`匹配。这或许有点难理解，但是一旦你习惯了这种用法，会发现它相当强大。这可以使我们通过约定的方式将 action 与 URL 松耦合。

在以上配置中，map 属性里指定了服务端点对于不同操作允许的 HTTP 请求类型：fetch 操作对应 GET 方法、edit 操作允许 PUT 方法，而 delete 操作只允许 DELETE 方法。通过这种方式，我们可以合理地控制应用的语义。你可能对余下的代码比较熟悉，即创建一个 Express 应用，并使用到下列两个插件：

- JSON 解析器
- Seneca Web 插件

到这里，我们已经完成了 Seneca 与 Express 的集成。如果想为 Seneca 添加新的可通过 API 提供服务的 action，唯一需要做的就是修改 map 属性，将 action 与相应的 HTTP 请求绑定。

虽然我们只构建了一个非常简单的微服务，但是这已经囊括了一个 CRUD（Create、Read、Update、Delete）应用中绝大部分的常见模式。此外，我们已经基于 Seneca 应用毫不费力地创建了一个小型 REST API。接下来，需要做的是配置基础设施（MongoDB），以及准备进行微服务的部署。

邮件服务：一个常见的问题

每个公司都需要邮件服务，我们可以通过邮件给客户发送通知、账单以及注册邮件。在我曾经工作过的公司中，邮件服务经常出现以下问题：邮件投递失败、邮件重复发送、邮件投递错误等。令人生畏的是，发送邮件这样简单的功能管理起来是如此复杂。

总而言之，邮件服务是我们编写一个微服务时的首选，考虑到它有以下特点：

- 只处理一个功能点。
- 处理得很好。
- 持有自己的专有数据。

邮件服务是康威定律在我们系统里无形中发挥作用的典型事例。我们根据公司现有的沟通模式来建模，从而设计系统。

如何发送邮件

回归邮件服务的本质，我们应该如何发送邮件？这里，我讨论的并不是选用何种网络协议来发送邮件，或者请求中可接受的最小 header，而是从业务角度上，讨论发送一封邮

件需要什么：

- 标题
- 内容
- 目标地址

以上三点是发送一封邮件必需的内容。如果进而讨论送达确认、邮件安全、密件抄送等，这就偏离了主题。然而，我们应该遵循精益方法论：初始构建一个最小的可行产品，不断地迭代改进，直到满足期望结果。

我接触的每个项目在邮件服务的选择上都非常有争议，最终采用的邮件服务都与系统本身紧耦合，导致平滑切换邮件服务相当困难。然而，微服务可以解决这个问题。

接口定义

正如我之前提到的，虽然邮件服务的实现看起来容易，但是企业邮件功能最后可能变得一团糟。因此，我们首先要明确最小需求：

- 如何渲染邮件？
  - 邮件渲染是否属于邮件操作的限界上下文？
  - 是否另起一个微服务负责邮件渲染？
  - 是否使用第三方工具管理邮件？
- 是否保留已发送邮件数据以备审计需要？

在这个微服务中，我们使用了 Mandrill，它支持企业级邮件发送、追溯已发送邮件和创建可在线编辑的邮件模板。

微服务的代码如下：

```
var plugin = function(options) {
  var seneca = this;
  /**
   * 使用模板发送邮件
   */
  seneca.add({area: "email", action: "send", template: "*"},
    function(args, done) {
// TODO: More code to come.
```

```
  });

  /**
   * 发送包含内容的邮件
   */
  seneca.add({area: "email", action: "send"}, function(args, done) {
// TODO: More code to come.
  });
};
```

我们有两种邮件发送模式：一种是使用模板，另一种是在发送请求中自带内容。

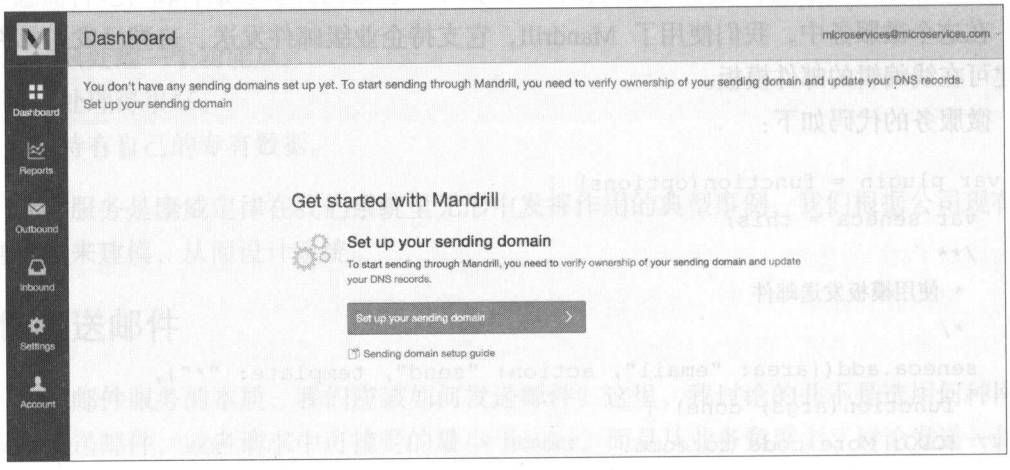
如你所见，我们定义的都是关于发送邮件的信息。其他使用邮件发送的微服务并不需要知道 Mandrill 的那些术语，唯一需要调用方感知的是对模板的处理。我们将模板的渲染委托给邮件发送方，但是这也并无大碍，甚至当不使用 Mandrill 之后，我们将会自己来实现对于内容的渲染。

随后我们会继续分析代码。

设置 Mandrill

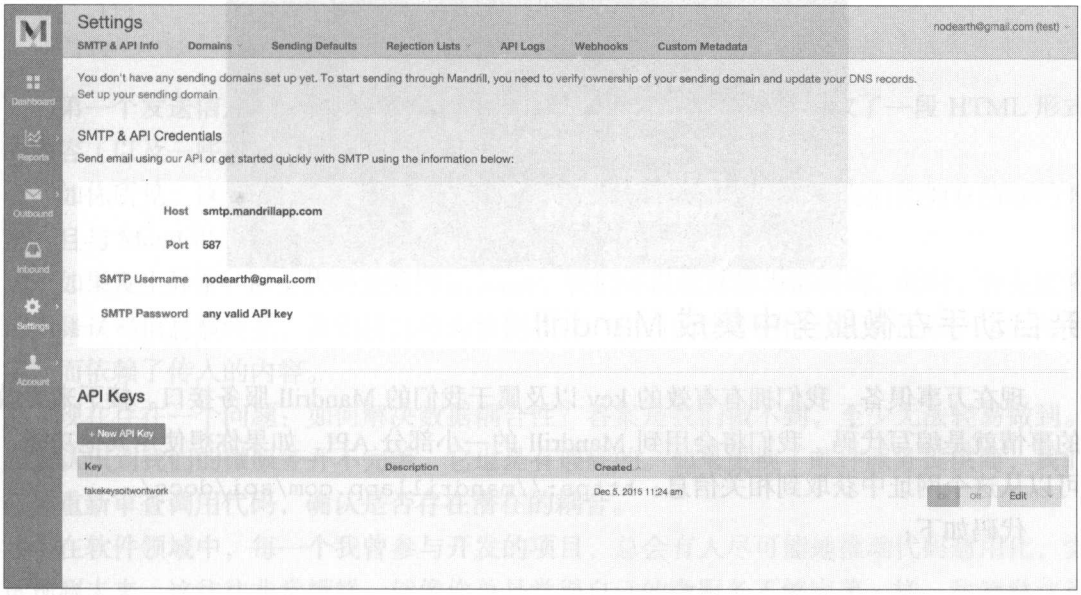
Mandrill 的使用相当容易，而且设置也没什么问题。然而我们将采用 Mandrill 的测试模式（test mode），从而确保邮件不会真正被发送，并且可以按照我们的需求访问 API。

首先，需要创建一个 Mandrill 账户。访问网址 `https://mandrillapp.com` 并用邮箱进行注册后，即可以进行访问，如下图所示：



现在已经创建了一个账户，可用该账户进入测试模式。单击右上角邮箱地址所在位置，开启菜单栏中的测试模式。Mandrill [译注2]左边栏会变为橘黄色。

接下来，我们将创建一个 API key，它是 Mandrill API 需要使用的登录信息。单击 Settings⇨SMTP & API Info，添加一个新的 key（别忘了勾选标记 key 为测试使用的复选框）。如下图所示：



你现在唯一需要的信息就是生成的 key。让我们对 API 进行如下测试：

```
var mandrill = require("mandrill-api/mandrill");
var mandrillClient = new mandrill.Mandrill("<YOUR-KEY-HERE>");

mandrillClient.users.info({}, function(result){
  console.log(result);
}, function(e){
  console.log(e);
});
```

只需要以上几行代码，就可以测试 Mandrill 是否已经启动并运行，以及 key 是否正确。这段代码将会得到与下图类似的 JSON 对象：

译注2 现在 Mandrill 已经收费，因此无法免费试用 Mandrill。

```
{ username: 'youremail@yourdomain.com',
  created_at: '2015-12-05 10:55:59.02874',
  public_id: 'yourpublicid',
  reputation: 33,
  hourly_quota: 25,
  backlog: 0,
  stats:
   { today:
      { sent: 0,
        hard_bounces: 0,
        soft_bounces: 0,
        rejects: 0,
        complaints: 0,
<...continues...>
```

亲自动手在微服务中集成 Mandrill

现在万事俱备，我们拥有有效的 key 以及属于我们的 Mandrill 服务接口。接下来要做的事情就是编写代码。我们将会用到 Mandrill 的一小部分 API，如果你想使用其他功能，可以从这个网址中获取到相关信息：`https://mandrillapp.com/api/docs/`。

代码如下：

```
/**
  * 发送包含内容的邮件
  */
 seneca.add({area: "email", action: "send"}, function(args, done)
  {
  console.log(args);
  var message = {
    "html": args.content,
    "subject": args.subject,
    "to": [{
      "email": args.to,
      "name": args.toName,
      "type": "to"
    }],
    "from_email": "info@micromerce.com",
    "from_name": "Micromerce"
```

```
    }
    mandrillClient.messages.send({"message": message},
      function(result) {
      done(null, {status: result.status});
    }, function(e) {
      done({code: e.name}, null);
    });
  });
```

第一个发送信息的方法并没有使用到模板。我们只是从应用中获取了一段 HTML 形式的内容（以及一些其他参数）并通过 Mandrill 完成发送。

如你所见，这个方法中仅有两个与外界的交互点：入参和返回。它们都拥有清晰的约定，且与 Mandrill 无关。那数据部分又如何呢？

如果发生异常，以上代码会返回 `e.name`，我们可以将其作为错误码。此时，分支流将因为*错误码*信息被终止，我们将这称为数据耦合。我们的软件组件并没有依赖于之前的约定，而依赖了传入的内容。

现在存在一个问题：如何解决数据耦合性？答案是我们做不到，至少无法轻易做到。必须认识到我们的微服务并不完美，它也是有瑕疵的。如果更换了邮件服务提供商，我们必须重新审查调用代码，确认是否存在潜在的耦合。

在软件领域中，每一个我曾参与开发的项目，总会有人尽可能地推动代码通用化，尝试预测未来，这往往非常糟糕，就像你总是觉得自己的微服务不够完美一样。我常常注意到：我们尽了很大的努力让项目变得完美，却忽略了自己可能失败并且无能为力的事实。在软件世界中失败是难免的，我们必须为此做好准备。

随后，我们将会看到一个将人类天性融入微服务的模式：*断路器*。

如果 Mandrill 因为未签名问题拒绝了你的邮件时，不要惊讶。这是因为它无法验证邮件发送域名（示例中的虚假域名是不存在的）。如果想确保 Mandrill 能够处理邮件（即使在测试模式下），只需加上域名验证配置。

在 Mandrill 的官方文档中可以查阅到更多信息：

`https://mandrillapp.com/api/docs/`

第二种发送邮件的方式是使用邮件模板。在这方面，Mandrill 提供了灵活的 API：

- 提供了逐收件人变量，以便于我们将邮件发送给一组客户。
- 拥有全局变量。
- 允许内容替换（可以替换模板中的所有部分）。

由于本书篇幅有限，为了方便起见，我们仅使用全局变量。

来看一下以下代码：

```
/**
 * 使用模板发送邮件
 */
seneca.add({area: "email", action: "send", template: "*"},
  function(args, done) {
  console.log("sending");
  var message = {
    "subject": args.subject,
    "to": [{
      "email": args.to,
      "name": args.toName,
      "type": "to"
    }],
    "from_email": "info@micromerce.com",
    "from_name": "Micromerce",
    "global_merge_vars": args.vars,
  }
  mandrillClient.messages.sendTemplate(
    {"template_name": args.template, "template_content": {},
      "message": message},
  function(result) {
    done(null, {status: result.status});
  }, function(e) {
    done({code: e.name}, null);
  });
});
```

现在，我们可以在 Mandrill 中创建自己的模板并使用它们发送邮件，同时你可以让其他人来管理模板。重申一遍，术业有专攻，我们的系统专职于邮件发送，你应该将创建邮件的工作交给别人（可以是那些知道如何与客户打交道的市场部成员）。

让我们对该微服务进行分析。

- **数据存储在本地：**因为数据存储在 Mandrill 上，所以严格意义上并不是本地，但是从设计的角度考虑，数据确实是保留在该服务内。
- **具有良好的内聚性：**仅负责发送邮件，并且专而精。
- **具有合适规模：**这很好理解，它没有不必要的抽象，并且可以很轻易地重写。

之前谈论到 SOLID 设计原则时，我们总是跳过里氏替换原则。其实该原则表示软件代码必须保持语义正确。例如，我们编写了一个操作某一抽象类的面向对象程序，那么该程序必须能够操作该抽象类的所有子类对象。

回到 Node.js，如果我们的服务能够处理简单的邮件发送，那么它应该易于扩展且能够在不改变已有功能的前提下加入新功能。从日常生产操作的角度考虑，如果你的系统添加了新功能，但到最后才考虑对已有功能进行重新测试，甚至直接交付生产，这可能会引入无人察觉的 bug。

考虑以下用例：我们想要给两位接收者发送相同的邮件。虽然 Mandrill API 提供了可直接调用的功能，但是之前并没有考虑过潜在的抄送需求。

因此，我们将会在 Seneca 中加入新的 action 来完成以上需求，代码如下：

```
/**
 * 发送一封包含内容的邮件
 */
seneca.add({area: "email", action: "send", cc: "*"},
  function(args, done) {
  var message = {
    "html": args.content,
    "subject": args.subject,
    "to": [{
      "email": args.to,
      "name": args.toName,
      "type": "to"
    },{
      "email": args.cc,
      "name": args.ccName,
      "type": "cc"
    }],
    "from_email": "info@micromerce.com",
```

```
        "from_name": "Micromerce"
      }
      mandrillClient.messages.send({"message": message},
        function(result) {
        done(null, {status: result.status});
      }, function(e) {
        done({code: e.name}, null);
      });
    });
```

我们为 Seneca 添加了一个方法，该方法将接收参数列表中包含 cc 的调用，并且使用发送 API 中 Mandrill 的邮件抄送功能发送邮件。如果我们需要使用该方法，必须修改以下调用代码的签名。

```
seneca.act({area: "email", action: "send", subject: "The Subject", to:
"test@test.com", toName: "Test Testingtong"}, function(err, result){
// More code here
});
```

签名修改如下：

```
seneca.act({area: "email", action: "send", subject: "The Subject",
to: "test@test.com", toName: "Test Testingtong", cc: "test2@test.com",
ccName: "Test 2"}, function(err, result){
// More code here
});
```

模式匹配会匹配最具体的输入项，因此，如果某个 action 比其他 action 匹配更多的参数项，那么该 action 将会被调用。

这是 Seneca 的一大亮点，我们将其称为 action 的多态。可以通过不同参数为同一个 action 定义不同版本，这些版本最终将执行略有差异的功能，并且如果我们能完全保证正确性就可以复用代码。记住，微服务推行无共享方式：代码重复可能不会像两个 action 耦合那么糟。

以下是邮件服务的 package.json 文件：

```
{
  "name": "emailing",
  "version": "1.0.0",
```

```
  "description": "Emailing sub-system",
  "main": "index.js",
  "keywords": [
  "microservices",
  "emailing"
  ],
  "author": "David Gonzalez",
  "license": "ISC",
  "dependencies": {
  "mandrill-api": "^1.0.45",
  "seneca": "^0.8.0"
  }
}
```

回退策略

当你设计一个系统时，通常应该考虑已有组件的可替换性。例如，当使用 Java 持久化技术时，我们倾向于使用标准接口（JPA），从而可以轻易地更换底层实现。

微服务采用了同样的方法，但是它们将问题进行隔离，而不是致力于简单的可替换性。正如之前的代码，在 Seneca 的 action 中，我们并没有隐藏使用 Mandrill 发送邮件的事实。

如同我之前提到的，邮件服务虽然表现上看起来很简单，但是最终会遇到许多问题。

假设我们想用 Gmail 这样的 SMTP 服务器来替代 Mandrill，不必做任何特殊处理，只需要更换底层实现，推出微服务的新版本即可。

切换过程相当简单，代码如下：

```
var nodemailer = require('nodemailer');
var seneca = require("seneca")();
var transporter = nodemailer.createTransport({
  service: 'Gmail',
  auth: {
    user: 'info@micromerce.com',
    pass: 'verysecurepassword'
  }
});

/**
```

```
 * 发送一封包含内容的邮件
 */
seneca.add({area: "email", action: "send"}, function(args, done) {
  var mailOptions = {
    from: 'Micromerce Info <info@micromerce.com>',
    to: args.to,
    subject: args.subject,
    html: args.body
  };
  transporter.sendMail(mailOptions, function(error, info){
    if(error){
      done({code: e}, null);
    }
    done(null, {status: "sent"});
  });
});
```

现在，对于外界而言，邮件发送服务的简化版本是采用 SMTP，通过 Gmail 进行邮件投递。

在本书的后续章节中，你将会看到，在微服务网络中为同一个接口交付新版本是相当容易的。只要我们遵循接口规范，那么就能够与具体的实现解耦。

我们甚至可以只在一个服务器上部署新版本，然后将部分流量引导到该服务器上，从而在不影响所有客户的情况下，验证新版本的正确性，换句话说，即验证是否会出现问题。

在本节中，我们看到了如何编写邮件发送服务。通过编写部分样例代码，我们演示了在业务上出现新需求时，或者底层服务商无法满足技术需求时，微服务如何快速做出响应。

订单管理服务

订单管理服务主要负责处理用户通过 UI 填写的订单。正如之前提到的，我们不会使用流行的可视化框架去建立一个复杂的单页面应用，因为这超出了本书的范畴。但是，为了之后能够构建前端，我们将会提供 JSON 接口。

订单管理服务引入了一个值得注意的问题：该微服务需要访问商品信息，例如商品名、价格、可售库存等。但是，商品信息存放在商品管理服务中，我们该如何处理？

当然，这个问题的答案看起来似乎很简单，实际上却需要进行一点思考。

根据如何获取非本地数据来定义微服务

我们的微服务需要完成以下三项工作：

- 获取订单
- 创建订单
- 删除已有订单

当获取订单时，我们选择的方式非常简单，即通过主键查询订单信息。我们可以再增加其他查询条件，比如价格、日期等。但是，我们只关注微服务本身，所以将保持该功能的简单性。

当删除已有订单时，选择的方案同样很明确：通过 ID 删除订单。同样的，我们可以选择更多删除条件，但是仍然会保持功能的简单性。

当我们创建订单时，会遇到问题。在我们的小型微服务架构中，创建一条订单将会给客户发送一封邮件，提示客户我们正在处理他们的订单，并附加上订单的细节，如下所示：

- 商品数目
- 商品单价
- 商品总价
- 订单 ID（客户可以通过订单 ID 来反馈问题）

如何获取商品详细信息？

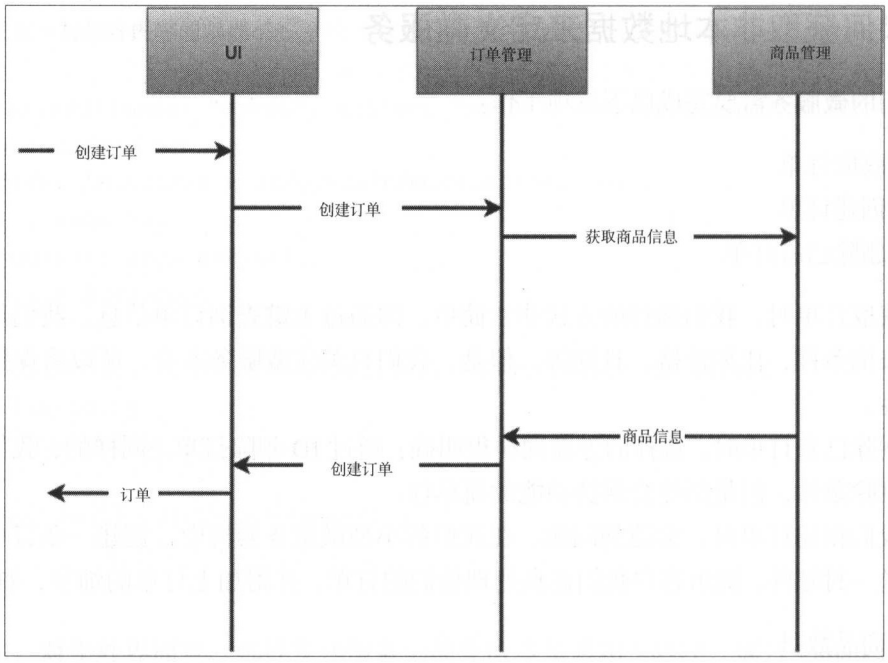
可以从本章开篇的微电商架构图中看到，订单服务仅仅被 UI 调用，其负责获取商品的名字、价格等。在此，我们可以采取以下两种策略：

- 订单管理服务调用商品管理服务，获取商品信息。
- UI 调用商品管理服务并且将数据委托给订单管理服务。

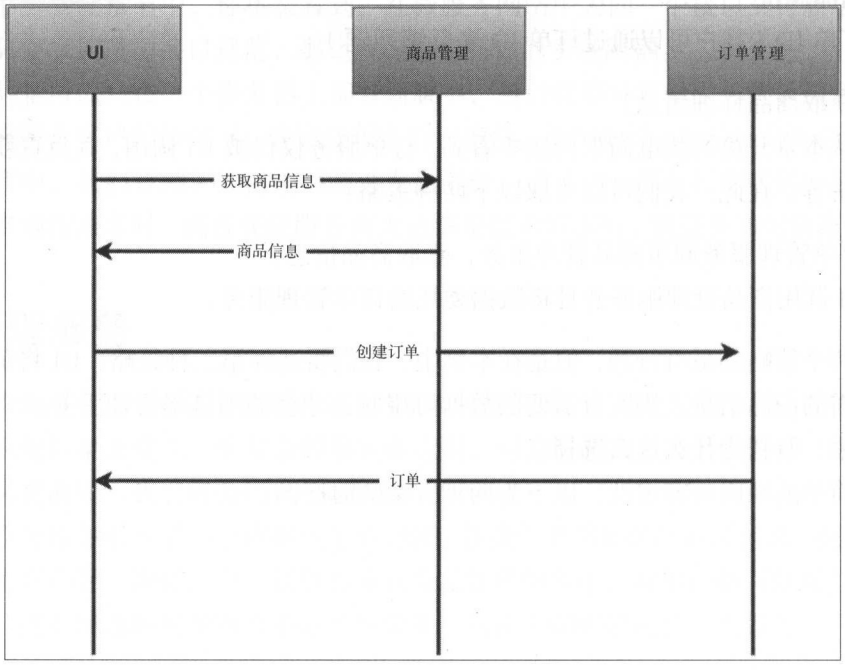
以上两个策略都是可行的，但是在本例中，我们将选择第二种策略：UI 将负责获取创建订单所需的商品信息，当所有需要的数据可用时，才会调用订单管理服务。

接下来，解释为什么这么选择。

一个简单的原因是容灾性。以下为两种方案的时序图：



第二种方案的时序图如下：



首先可以看出，以上两个策略最明显的区别在于调用的深度。在第一个例子中，有两层调用深度（UI 调用订单管理服务，订单管理服务再调用商品管理服务）。而在第二个例子中，只有一层调用深度。这对于我们的体系结构有很直接的影响，如下所述：

- 当遇到问题时，如果只有一层调用深度，那么我们就不需要在多处进行问题检查。
- 系统将更具弹性。当出现问题时，UI 将及时发现并返回相应的 HTTP 状态码，这样，在面向客户的微服务系统中，不同层次的错误就不需要再进行转换。
- 第二种方案使部署与测试也更加简单。虽然并没有简单太多，但是我们可以不需要订单管理服务，直接判断 UI 是否成功调用了商品管理服务。

在本例中我们选用了方案二，不选用具有两层调用深度的架构，其他的场景也应如此：在你设计面向微服务的架构时，必须优先考虑网络拓扑，因为这个是难以改动的层面之一。

在某些情况下，如果追求极致的灵活性，可以使用具有发布者/订阅者功能的消息队列，这样，微服务可以订阅不同类型的消息，并且将消息发送给不同的服务进行消费。但是，为了避免单点失败，消息队列的架构会比较复杂。

订单管理服务代码

让我们来看一下订单管理的代码：

```
var plugin = function(options) {
  var seneca = this;

  seneca.add({area: "orders", action: "fetch"}, function(args,
    done) {
    var orders = this.make("orders");
    orders.list$({id: args.id}, done);
  });

  seneca.add({area: "orders", action: "delete"}, function(args,
    done) {
    var orders = this.make("orders");
    orders.remove$({id: args.id}, function(err) {
        done(err, null);
    });
  });
```

```
}
module.exports = plugin;
```

如你所见，以上代码并没有任何复杂性，唯一值得注意的点是缺少创建订单的 action 代码。

调用远端服务

到目前为止，我们假设所有的微服务都运行在同一台机器上，但是事实远没有这么理想。在实际生产中，微服务是分布式的，我们需要使用合适的传输协议来进行服务器间的消息通信。

nearForm 是开发 Seneca 的公司，它已经为我们处理了这些问题，并且开源了相关代码。

作为一个模块化系统，Seneca 采用了插件体系。Seneca 默认加载了一个捆绑插件，其使用 TCP 作为传输协议，但是创建一个新的传输插件也并不困难。

在编写本书时，我编写了一个传输插件，代码托管在：

https://github.com/dgonzalez/seneca-nservicebus-transport/

使用该插件，可以通过 NServiceBus（基于.NET 的企业总线）来进行 Seneca 的信息路由，同时更改客户端与服务端的配置。

以下是如何将 Seneca 指向不同机器的示例：

```
var senecaEmailer = require("seneca")().client({host: "192.168.0.2",
port: 8080});
```

如同第 2 章中提及的，Seneca 会默认加载 `tcp` 传输插件。我们指定其指向主机 `192.168.0.2` 的 `8080` 端口。

只需要这么简单的处理，从现在开始，当我们对 `senecaEmailer` 执行 act 命令时，传输插件会将消息发送给邮件服务并接收响应。

让我们看一下余下的代码：

```
seneca.add({area: "orders", action: "create"}, function(args,
  done) {
  var products = args.products;
  var total = 0.0;
```

```
    products.forEach(function(product){
      total += product.price;
    });
    var orders = this.make("orders");
    orders.total = total;
    orders.customer_email = args.email;
    orders.customer_name = args.name;
    orders.save$(function(err, order) {
      var pattern = {
        area: "email",
        action: "send",
        template: "new_order",
        to: args.email,
        toName: args.name,
        vars: {
          // ... 模板渲染所需的变量，包括products  ...
        }
      }
      senecaEmailer.act(pattern, done);
    });
  });
```

如你所见，我们获取到了包含全部所需信息的商品列表，并传递给邮件服务进行渲染。

如果邮件服务的主机发生了改变，只需要修改 `senecaEmailer` 变量中的配置信息即可。

即使改变数据传输渠道，例如，编写一个通过 Twitter 进行数据传输的插件，我们的传输插件仍然能处理特定的需求，从而保持对应用的透明。

弹性胜于完美

在之前的例子中，我们构建了一个能够调用其他微服务并处理返回结果的微服务。然而，我们还需要考虑以下几点：

- 如果邮件服务宕机了会发生什么？
- 如果配置了错误的邮件服务端口，会发生什么？

我们可以抛出无数诸如此类的“如果”，几页也写不完。

人类并不是完美的，因此他们构建的产物也同样是不完美的，软件系统当然也不例外。

更糟糕的是，人类并不善于在逻辑流中发现潜在问题，软件系统由于其复杂性，使得发现潜在问题更加困难。

在其他编程语言中，处理异常是非常普通的一种操作，但是在 JavaScript 中，异常却是重中之重：

- 如果 Java Web 应用抛出异常，它会杀死本次调用栈，然后 Tomcat 或者其他 Web 容器会将错误信息返回给客户端。
- 如果 Node.js 应用抛出异常，由于应用是单线程的，整个应用都将会被杀死。

如你所见，Node.js 中许多回调函数的第一个参数都是 error。

当提及微服务时，这个 error 至关重要，因为我们需要保证系统的弹性。实际上，邮件发送失败并不代表整个订单处理流程的失败，而且，随后邮件还可以被重新手动发送。这就是所谓的最终一致性。我们必须考虑到系统可能会因为某种原因宕机这一情况。

在这种情况下，如果邮件发送出现问题，而且我们可能已经将订单信息存入数据库，调用方即本例中的 UI，那么需要有足够的信息来判断客户是否应当收到严重错误提示消息，或者只收到如下告警：

> “您的订单处于待处理状态，订单详情会在两个工作日内发送给您。感谢您的耐心等待。”

通常，在实际应用中，就算无法处理某个请求，应用系统仍然会继续工作，这更多是出于业务角度考虑，而非技术角度。这里有一个很关键的细节：当我们构建微服务时，康威定律时刻影响着我们这些开发人员，为已有的业务场景构建模型时，由于人类不完美的天性，并不能保证构建完全成功。如果你暂时有一个无法完成的任务，请在印象笔记或者其他类似工具中记录下来，当解决了该任务的阻碍后，再回过头来完成这个任务。

上一条告警内容明显比以下提示内容读起来更友好：

> “某些事情的发生导致了某些问题，但是我们无法给予你更多问题详情。”

当我访问一些网页失败时，经常会得到类似的提醒。

我们将这种处理错误的方式称为系统降级：虽然不能具备 100%的功能，但是在少量功能不可用的情况下，系统仍然可以继续运行，而不是直接失败。

请认真思考一下，在你的大型企业级应用系统中，由于无法访问非核心的第三方服务，导致过多少次 Web 服务调用发生事务回滚？

在本节中，我们构建了一个微服务，其依赖于其他微服务来处理用户的请求：订单服务依赖于邮件服务来完成整个请求。接着，我们讨论了服务的弹性，以及系统架构中具有弹性对于提供最佳服务的至关重要性。

UI——API 聚合的产物

到目前为止，我们构建了几个各自独立的微服务。它们各司其职，负责处理系统中某个特定功能，例如邮件发送、商品管理、订单处理。现在，我们准备构建一个微服务，仅用于促进上述微服务间的通信。

现在，我们准备构建一个与其他服务交互的微服务，这也是直接面向客户的前端服务。

当我最初组织本章内容时，这个微服务并不在讨论范围内。但是，经过深思熟虑，与其堆砌有关 API 聚合的概念，不如通过构建前端微服务的方式来讲解。

前端微服务的必要性

首先考虑可扩展性。当处理 HTTP 流量时，流量是呈金字塔形的，前端的流量会多于后端。通常，为了流量能够成功传递到后端，前端服务需要处理以下请求：

- 获取表单
- 验证合法性
- 管理 PRG 模式[译注3]（`https://en.wikipedia.org/wiki/Post/Redirect/Get`）

如你所见，前端需要处理相当多的逻辑，因此当软件系统繁忙时，很容易发生容量问题。如果我们能够正确地使用微服务，那么只需几次单击或者几条命令即可触发系统进行自动伸缩。

代码

直到现在，我们都是在单机上测试代码。从测试角度上看，这并没有什么问题，但是，我们正在构建微服务，它们实际上应该是分布式的。因此，我们需要在 Seneca 中配置各个服务的访问地址：

[译注3] PRG 是指 Post/Redirect/Get，这种模式可以防止用户刷新页面时造成表单重复提交。

```
var senecaEmailer = require("seneca")().client({
  host: "192.168.0.2",
  port: 8080
});
var senecaProductManager = require("seneca")().client({
  host: "192.168.0.3",
  port: 8080
});
var senecaOrderProcessor = require("seneca")().client({
  host: "192.168.0.4",
  port: 8080
});
```

在以上代码中创建了三个 Seneca 实例。它们就像各个服务之间的通信管道。

让我们解释一下以上代码：

Seneca 默认使用 TCP 插件进行数据传输。这意味着，Seneca 将监听服务器的 URL 地址/act。举例来说，当创建 senecaEmailer 时，Seneca 指向的 URL 地址为 http://192.168.0.2:8080/act。

实际上，我们可以通过 curl 来验证，如果执行以下命令行，用有效的 Seneca 命令替换 <valid Seneca pattern>，将会获取到服务器返回的 JSON 格式的内容，它是 done 函数的第二个参数。

```
curl -d '<valid Seneca pattern>' -v http://192.168.0.2:8080/act
```

Seneca 默认使用 TCP 插件进行数据传输，因此，如果没有特别指定，Seneca 将使用 TCP 插件访问其他服务器和监听调用。

我们来看一个简单的例子：

```
var seneca = require("seneca")();
seneca.add({cmd: "test"}, function(args, done) {
  done(null, {response: "Hello World!"});
});

seneca.listen({port: 3000});
```

如果我们执行以上程序，将会在终端中得到如下输出：

```
2015-12-14T01:23:48.944Z asrwzxwx2u4e/1450920228931/7488/- INFO hello Seneca/0.8.0/asrwzxwx2u4e/1450920228931/7488/-
2015-12-14T01:23:49.102Z asrwzxwx2u4e/1450920228931/7488/- INFO listen {port:3000}
```

这表明 Seneca 正在监听 3000 端口，可以通过以下方式进行测试：

```
curl -d '{"cmd": "test"}' -v http://127.0.0.1:3000/act
```

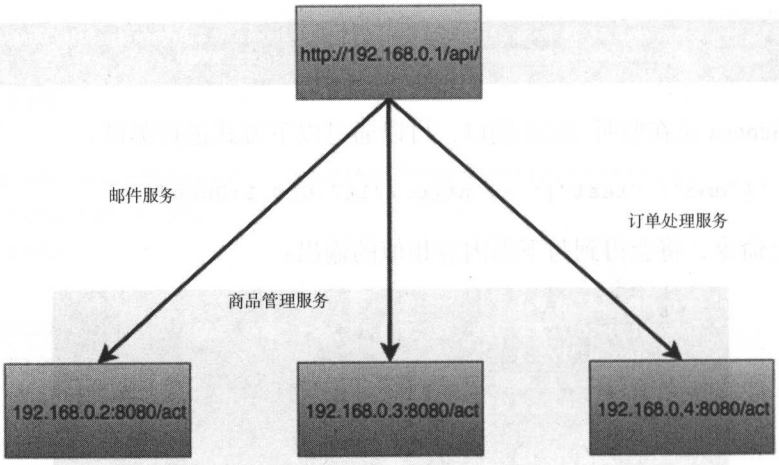
执行以上命令，将会得到与下图内容相似的输出：

```
*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> POST /act HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 15
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 15 out of 15 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json
< Cache-Control: private, max-age=0, no-cache, no-store
< Content-Length: 27
< seneca-id: i45q1ayb0wl1
< seneca-kind: res
< seneca-origin: curl/7.43.0
< seneca-accept: asrwzxwx2u4e/1450920228931/7488/-
< seneca-track:
< seneca-time-client-sent: 0
< seneca-time-listen-recv: 0
< seneca-time-listen-sent: 0
< Date: Thu, 14 Dec 2015 01:26:15 GMT
< Connection: keep-alive
<
* Connection #0 to host 127.0.0.1 left intact
{"response":"Hello World!"}%
```

上图是我们的终端与 Seneca 服务器进行 TCP/IP 会话的内容，最后一行是返回的结果。

因此，之前创建的三个 Seneca 实例组成了我们的微服务网络，Seneca 负责网络间的数据传输。

下图展示了单个 API 如何隐蔽后端的多个 Seneca 微服务（它们的本质是不同的 Seneca 实例）：



现在，让我们看看微服务的代码骨架：

```
var express = require("express");
var bodyParser = require('body-parser');
var senecaEmailer = require("seneca")().client({
  host: "192.168.0.2",
  port: 8080
});
var senecaProductManager = require("seneca")().client({
  host: "192.168.0.3",
  port: 8080
});
var senecaOrderProcessor = require("seneca")().client({
  host: "192.168.0.4",
  port: 8080
});

function api(options) {
  var seneca = this;

  /**
   * 获取所有商品列表
   */
  seneca.add({area: "ui", action: "products"}, function(args,
    done) {
```

```
    // 具体内容稍后讨论
  });
  /**
   * 根据id获取商品信息
   */
  seneca.add({area: "ui", action: "productbyid"}, function(args,
    done) {
    // 具体内容稍后讨论
  });

  /**
   * 新建订单
   */
  seneca.add({area: "ui", action: "createorder"}, function(args,
    done) {
    // 具体内容稍后讨论
  });

  this.add("init:api", function(msg, respond){
    seneca.act('role:web',{ use: {
      prefix: '/api',
      pin:  'area:ui,action:*',
      map: {
        products:  {GET:true}
        productbyid: {GET:true, suffix:'/:id'}
        createorder: {POST:true}
      }
    }}, respond)
  });
}
module.exports = api;
var seneca = require("seneca")();
seneca.use(api);

var app = require("express")();
app.use( require("body-parser").json());
app.use(seneca.export("web"));
```

```
app.listen(3000);
```

调用其他微服务的功能我们稍后再做讨论，先把注意力放在代码组织脉络上：

- 我们创建了一个新插件，名为 `api`，该插件的包装函数也命名为 `api`。
- 新插件需要处理以下三个 action：
  - 获取所有商品信息列表
  - 通过 ID 获取商品
  - 创建订单
- 这三个 action 将会调用两个不同的微服务：商品管理与订单管理。之后，我们会继续讨论这个话题。

Seneca 可以平滑地与 Express 集成并以此来提供 Web 能力。

到目前为止，万事俱备，但是插件的初始化函数在哪里？
乍看之下，这有点像黑魔法：

```
this.add("init:api", function(msg, respond){
  seneca.act('role:web',{ use: {
    prefix: '/api',
    pin:  'area:ui,action:*',
    map: {
      products:  {GET:true}
      productbyid: {GET:true, suffix:'/:id'}
      createorder: {POST:true}
    }
  }}, respond)
});
```

现在，解释一下以上代码：

1. Seneca 将会调用 `init:<plugin-name>` action 来初始化插件。
2. 通过参数 `prefix` 指定该插件监听以 `/api` 为前缀的 URL 地址。
3. 我们指定 Seneca 建立 URL 到 action 的映射关系，在这种情况下，所有的 `seneca.add(...)` 方法都必须包含一个值为 `ui` 的 `area` 参数。同时，我们要求

Seneca 对所有包含 action 参数的调用进行路由，忽略没有指定 action 参数的调用。此外，我们并不关注 action 参数的具体值，所以上面代码中使用“*”。

以下参数指定了模式匹配中可用的方法。然而，参数匹配是如何完成的？

参数 area 是显式的，因为我们已经将其值指定为 ui。

action 参数必须存在。

URL 必须以/api 为前缀。

综上所述，/api/products 与 action{area: "ui", action: "products"}关联。同样的，/api/createorder 与 action{area: "ui", action:"createorder"}关联。

其中参数 Productbyid 略显特殊。

Seneca 的关键词 pin 保证了调用代码拥有一对参数值，使得代码更易理解。但请注意，不明确的值将影响代码可读性。

现在，上面那些复杂的代码看起来更好理解了。

让我们重新回到 Seneca 中负责具体功能的 action 代码：

```
/**
 * 获取全部商品信息列表
 */
seneca.add({area: "ui", action: "products"}, function(args,
  done) {
  senecaProductManager.act({area: "product", action: "fetch"},
    function(err, result) {
    done(err, result);
  });
});

/**
 * 根据id获取商品信息
 */
seneca.add({area: "ui", action: "productbyid"}, function(args,
  done) {
  senecaProductManager.act({area: "product", action: "fetch",
    criteria: "byId", id: args.id}, function(err, result) {
```

```
        done(err, result);
      });
    });

    /**
     * 创建购买单件商品的订单
     */
    seneca.add({area: "ui", action: "createorder"}, function(args,
      done) {
      senecaProductManager.act({area: "product", action: "fetch",
        criteria: "byId", id: args.id}, function(err, product) {
        if(err) done(err, null);
        senecaOrderProcessor.act(area: "orders", action: "create",
          products: [product], email: args.email, name: args.name,
            function(err, order) {
          done(err, order);
        });
      });
    });
```

注意，在本章编写的服务程序中，为了纯粹地介绍服务设计理念，所以没有进行数据验证。在实际生产中，应该对所有来自非可信系统的输入数据进行验证，例如用户输入信息。

实际上，我们已经使用了前几个章节中讨论的全部知识，而且对 Seneca 在语义上有了进一步的认识。

我们创建了一个只有小部分功能的 API，但是通过这些功能，将几个不同的微服务功能进行了整合。

有一个值得注意的细节是，在创建订单的 action 中包含了大量嵌套调用。因此，为了简化代码，我们只创建仅含一个商品的订单。但是，如果我们为了等待非阻塞 action 的回调而嵌套太多调用，最终将得到金字塔形的代码，这将使得程序难以阅读。

这个问题的解决方案是：重构数据获取方式，并且/或者重新组织匿名函数，避免内联。

这个问题的另一个解决方案是：使用 promises 库，例如 Q 或者 Bluebird（`http://bluebirdsjs.com/`），这些库使我们可以使用 promises 将方法流串联起来：

```
myFunction().then(function() {
// 函数代码
}).then(function(){
// 函数代码
}).catch(function(error){
// 处理错误
});
```

通过使用这种方式替代一系列的回调，我们可以很好地将方法调用串联起来，并且添加了错误处理，避免异常“冒泡”。

如你所见，我们将 UI 作为除邮件系统外所有微服务的通信中心，并且有充分的理由这么做。

服务降级——当出现非灾难性故障时

我们用几百行易于理解的代码编写了一个小型系统，证明了微服务的强大。

除此之外，微服务在故障处理方面也很优秀。

- 如果邮件微服务出现故障会发生什么？
- 如果订单处理模块出现故障会发生什么？
- 我们能否从故障场景中恢复？
- 出现故障时客户会看到什么？

在单块系统中，这些问题毫无意义。邮件模块将是整体应用的一部分。除非进行特殊处理，否则发送邮件失败意味着一次常规错误。订单处理模块也是如此。

但是，在面向微服务的架构中又如何呢？

实际上，邮件服务发送部分邮件失败，甚至客户根本收不到邮件，这都不影响订单处理流程。我们将这种情况称为，性能降级或服务降级。系统的响应速度可能降低，但是部分功能仍然可以使用。

服务降级是系统某一特性失效时规避故障的一种能力。

那么订单管理服务发生故障又如何处理呢？我们能保证商品的相关调用仍然可用，但是不能再处理任何订单，这看起来也未尝不是一件好事。

订单管理服务代替 UI 服务触发邮件发送并不是巧合，我们只有在确定消费流程成功时，才会发送邮件，在其他情况下并不需要发送邮件。

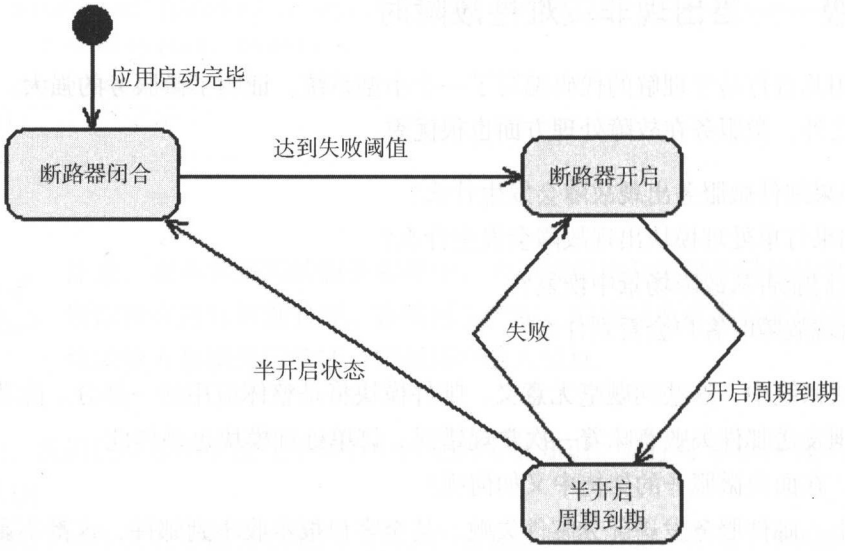
断路器

在上一小节中，我们讨论了当出现故障时，系统的降级处理，但是每个具有多年 IT 从业经验的人都知道，在绝大多数情况下，系统不会在出现故障时立刻失败。

最常见的情况就是超时。在某个时间段内，服务器繁忙，导致请求失败，对客户造成糟糕的用户体验。

如何解决这一特有问题呢?

我们可以通过引入断路器来解决这个问题，如下图所示：



断路器可以防止请求发送到不稳定的服务器上，避免应用失常的情况发生。

如上图所示，断路器有以下三种状态。

- **关闭**：断路器闭合，请求可以发往其目标服务器。
- **开启**：断路器开启，请求无法通过断路器，客户端会得到错误提示。经过一段时间，系统会重试通信。

- **半开启：**断路器对服务进行重试，如果可以正常连接，则可以继续向该服务发送请求，断路器闭合。

通过这一简单的机制，可以防止错误扩散到整个系统，避免“雪崩效应”。

理想情况下，断路器应该是异步的。这意味着即使没有请求，系统应该每隔数秒或者数毫秒重新尝试连接故障服务，以继续进行正常操作。

出错是人类的天性：最好能未雨绸缪。

断路器还可以给技术支持工程师发送告警。鉴于我们系统的特性，某个服务不可访问可能会带来严重的问题。你是否能够想象，银行系统无法访问短信服务，从而导致不能发送双重认证码的场景？无论我们多么努力避免，这些问题总是会在某些情况下发生。因此，我们需要为故障做好准备。

微服务界大名鼎鼎的 Martin Fowler 写过一篇关于断路器的启发性文章，网址为 `http://martinfowler.com/bliki/CircuitBreaker.html`。

Seneca——一块使我们工作变得更容易的拼图

Seneca 相当棒。它使得开发者在想到一个小点子时，就能够将其转化为实际代码，而不是只停留在设想阶段。一个 action 具有明确的输入，并且为你提供了能够通过回调进行响应的接口。

曾经有多少次，你发现你的团队为了以合理的方式复用代码，从而纠结于一项应用的类结构？

Seneca 追求简洁性。我们并不是在对对象进行建模，而是在对系统中的某些部分建模，这让我们的工作变得更加容易。系统中的这些部分使用了对于对象而言极具内聚性和幂等性的代码。

Seneca 使得我们的工作更加轻松的另一种方式是插件化。如果你回顾本书中我们写过的代码，首先就会发现插件使用起来是多么方便。

插件将一批彼此相关的 action 进行了合理封装（这是否看起来和类很相似）。

我总是极力避免“过度工程化”的解决方案。开发人员极易陷入过早抽象化，编写一

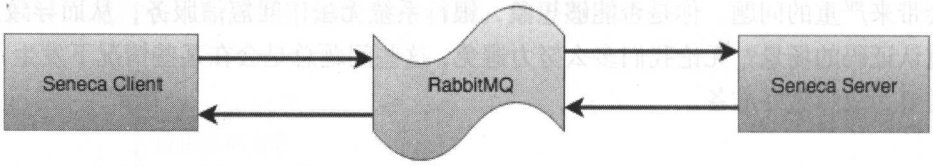
些不知道未来能否适用于大部分场景的代码。

我们没有意识到，对于被过度设计且每次变更都需要测试的功能，我们投入了太多精力去维护它们。

Seneca 不鼓励且避免这类设计方式。请把 Seneca 想象成一张小纸片例如便利贴，你需要在上面记录上周发生了什么。你应该知道该写什么合适，如果单张便利贴中的内容过于紧凑，还需要拆分到另一张便利贴上。

Seneca 还有一个优点就是可配置。正如我们之前看到的，Seneca 在数据存储与传输功能上做了一定的整合。

传输协议是 Seneca 的一个重要组成部分。到目前为止，我们知道 Seneca 的默认传输方式是通过 TCP 传输，但是是否可以使用消息队列呢？结构如下所示：



是的，我们可以这么做。这种方式已经被实现且在维护中了。

下列网址是 Seneca 用于将 HTTP 替换成 RabbitMQ 进行消息传递的插件：

https://github.com/senecajs/seneca-rabbitmq-transport

这份插件代码看起来貌似很复杂，但实际并非如此，如果你仔细阅读它，很快就能发现奥秘所在。

```
    seneca.add({role: 'transport', hook: 'listen', type: 'rabbitmq'},
hook_listen_rabbitmq)
    seneca.add({role: 'transport', hook: 'client', type: 'rabbitmq'},
hook_client_rabbitmq)
```

Seneca 将消息传输委托给 action，尽管看起来有些递归的意思，但这种方式真的棒极了！

一旦理解了 Seneca 和被选中的传输协议的工作方式，你立刻能够为 Seneca 编写传输插件。

当我为了编写本书开始学习 Seneca 时，我同样编写了使用 NServiceBus 的传输插件，地址为 http://particular.net/。

NServiceBus 是一个有趣的想法，通过它，你可以与许多存储和兼容 AMQP 的系统建立连接，并且作为客户端使用它们。比如，我们可以在一张 SQL Server 表中写入消息，一旦它们通过 NServiceBus，便可以从一个队列中消费它们。此外，还可以对历史消息进行直接审计。

借助于它的灵活性，我们几乎可以编写使用任意传输协议的插件。

Seneca 和 promise

前面几个章节中的所有代码都依赖于回调。只要你的代码嵌套层次小于等于三层，使用回调就是有利的。

然而，还有一种更好的管理 JavaScript 异步特性的方式：promise。

请看以下代码：

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>promise demo</title>
<script src="https://code.jquery.com/jquery-1.10.2.js"></script>
</head>
<body>

<button>Go</button>
<p>Ready...</p>
<div></div>
<div></div>
<div></div>
<div></div>

<script>
var effect = function() {
```

```
  return $( "div" ).fadeIn( 800 ).delay( 1200 ).fadeOut();
};

$( "button" ).on( "click", function() {
  $( "p" ).append( " Started... " );

  $.when( effect() ).done(function() {
  $( "p" ).append( " Finished! " );
  });
});
</script>

</body>
</html>
```

以上代码是 jQuery 中使用 promise 的用例片段。

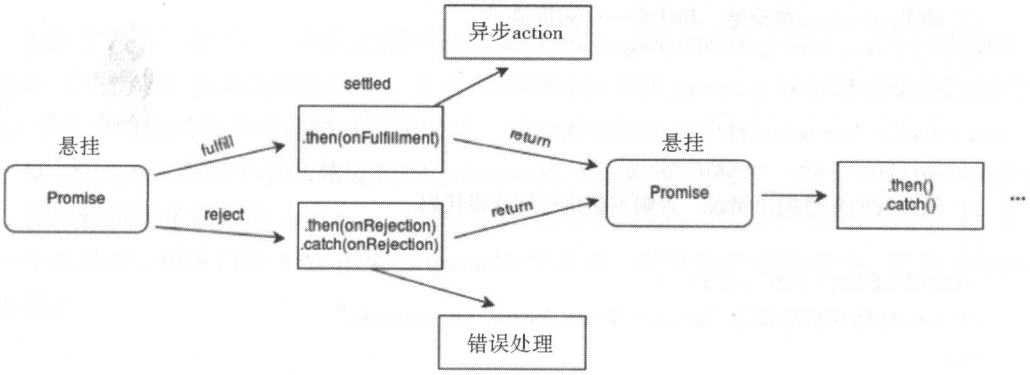
promise 这个词的本义如下：

> 声明或保证某人会做某事，或者某个特定事件会发生。

这正是 promise 的作用。在以上代码中，`$.when` 返回一个 promise。我们并不知道 effect 函数会执行多久，但是可以保证一旦它执行完毕，`done` 方法中的函数将会执行。这看起来很像回调，但是请看如下代码：

```
callhttp(url1, data1).then(function(result1) {
  // 此处result1可用
  return callhttp(url2, data2);
}).then(function(result2) {
  // 此处result2可用
  return callhttp(url3, data3);
}).then(function(result3) {
  // 所有结果完成，result3是最终结果
});
```

请不要尝试执行上述代码，它只是一个假设的例子，我们做的工作是将 promise 串联起来。通过 promise 链，使得代码垂直化，而不是形成难以阅读的金字塔形，如下图所示：



Seneca 默认并不是面向 promise 的框架，但是（凡事都有但是）通过使用 JavaScript 知名的 promise 库 Bluebird，我们可以对 Seneca 进行 promise 化，如下所示：

```
var Promise = require('bluebird');
var seneca = require('seneca')();

// 对.act()方法进行promise化，想要学习更多关于这个技术的知识可以参看：
// http://bluebirdjs.com/docs/features.html#promisification-on-steroids
var act = Promise.promisify(seneca.act, seneca);

// 返回一条证明promise完成的成功信息。
seneca.add({cmd: 'resolve'}, function (args, done) {
  done(null, {message: "Yay, I've been resolved!"});
});

// 返回一个错误强制拒绝 promise
seneca.add({cmd: 'reject'}, function (args, done) {
  done(new Error("D'oh! I've been rejected."));
});

// 使用了promise而不是回调方式的act()方法
act({cmd: 'resolve'})
  .then(function (result) {
  // 如果执行成功，结果为{message: "Yay, I've been resolved!"}
  })
  .catch(function (err) {
```

```
  // 如果promise被拒绝，和往常一样捕捉错误
  });

act({cmd: 'reject'})
  .then(function (result) {
  // 除非我们刻意抛出错误，否则不会执行到这段代码。
  })
  .catch(function (err) {
  // err的错误信息置为“D'oh! I've been rejected.”
  });
```

在以上代码中有两个重要的细节：

```
var act = Promise.promisify(seneca.act, seneca);
```

代码中使用了 promise 版本的 act 函数，如下所示：

```
act({cmd: 'reject'})
  .then(function (result) {
  // 除非我们刻意抛出错误，否则不会执行到这段代码。
  })
  .catch(function (err) {
  // err的错误信息置为“D'oh! I've been rejected.”
  });
```

在最后一块代码段中，有一个重要的细节，其并没有使用参数为 error 和 results 的回调函数，而是将以下两个方法进行了串联。

- Then：当 promise 完成时，执行其中的代码。
- Catch：当 promise 发生错误时，执行其中的代码。

我们可以按照这种结构方式编写出如下代码：

```
act({cmd: 'timeout'})
  .then(function (result) {
  // 只有当gate executor超时才会执行此方法。
  })
  .catch(function (err) {
  // err将被置为由gate executor抛出的超时错误
  });
```

这段代码中处理了一个我们之前从未提及的问题：gate executor 超时。这个问题发生在 Seneca 无法访问目标地址的情况下，并且能够通过前文的 promise 机制轻易地进行处理。then 方法中的代码除了处理超时错误以外，其他情况都不会执行。

JavaScript 生态圈中有几款完善的 promise 库可供选择。现在，由于 Bluebird 的简洁性，我比较倾向于使用它（https://github.com/petkaantonov/bluebird）。Q 作为另一个可选项，被流行的 SPA 框架 AngularJS 所采用，但是在日常使用中，它比 Bluebird 更加复杂。

调试

调试 Node.js 应用与调试其他应用类似，许多 IDE 譬如 WebStorm、IntelliJ 等都提供了传统的调试器，你能够设置断点，应用执行时会停止在特定的代码行上。

如果你能购买一款 IDE 的许可证书当然是最好的，但是对于 Google Chrome 用户来说，通过使用 node-inspector 这一免费的替代方案也能达到相似的效果。node-inspector 是一个 npm 包，它使得 Chrome 可以调试 Node.js 应用。

让我们看一下它是如何工作的：

1. 首先，我们需要安装 node-inspector：

   ```
   npm install -g node-inspector
   ```

   命令执行后，系统中添加了一条名为 node-inspector 的可用命令。如果执行该命令，会得到如下输出：

   ```
   ➜ code  node-inspector
   Node Inspector v0.12.7
   Visit http://127.0.0.1:8080/?port=5858 to start debugging.
   ```

   这表示调试服务已经开启。

2. 现在，我们要为应用打上特定标记，表示它需要被调试。

   我们将一个简单的 Seneca act 作为示例：

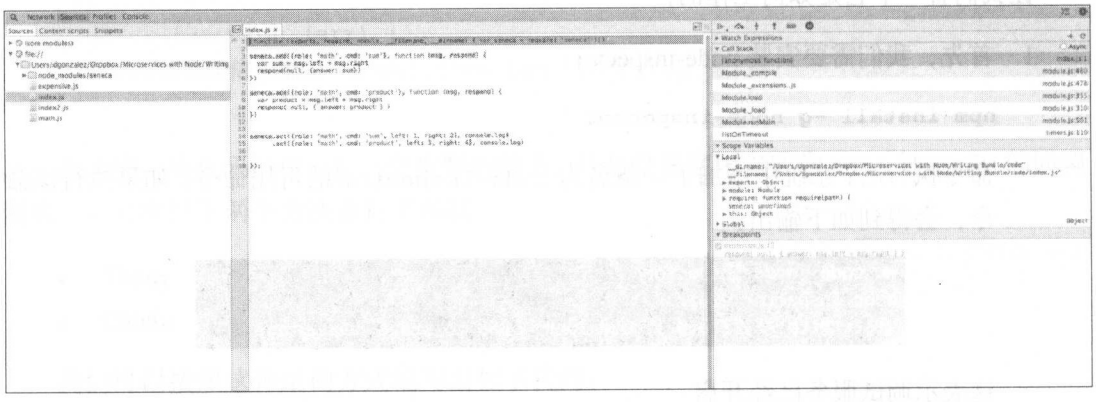
   ```
   var seneca = require( 'seneca' )()
   seneca.add({role: 'math', cmd: 'sum'}, function (msg,
     respond) {
     var sum = msg.left + msg.right
   ```

```
    respond(null, {answer: sum})
})

seneca.add({role: 'math', cmd: 'product'}, function (msg,
  respond) {
  var product = msg.left * msg.right
  respond( null, { answer: product } )
})


seneca.act({role: 'math', cmd: 'sum', left: 1, right: 2},
  console.log)
seneca.act({role: 'math', cmd: 'product', left: 3, right: 4},
  console.log)
```

3. 现在，执行以下命令进入调试模式：

```
node index.js --debug-brk
```

通过访问 URL 地址 `http://127.0.0.1:8080/?port=5858` 可以使用调试器。

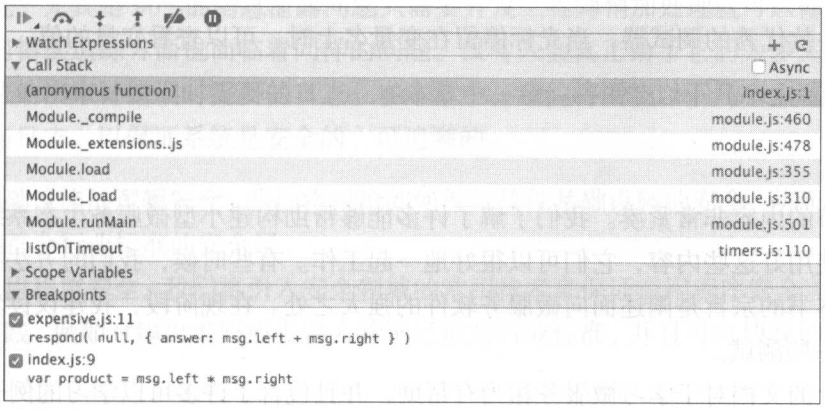


我确信上图中的内容对于世界上每个开发者来说都非常熟悉：这展示了代码的 Chrome 调试器。你可以看见第一行被标为蓝色高亮，应用程序停止在了第一行代码上，此时可以通过单击行号设置断点，如下图所示：

index.js ×

```
1  (function (exports, require, module, __filename, __dirname) { var seneca = require( 'seneca' )()
2
3  seneca.add({role: 'math', cmd: 'sum'}, function (msg, respond) {
4    var sum = msg.left + msg.right
5    respond(null, {answer: sum})
6  })
7
8  seneca.add({role: 'math', cmd: 'product'}, function (msg, respond) {
9    var product = msg.left * msg.right
10   respond( null, { answer: product } )
11 })
12
13
14 seneca.act({role: 'math', cmd: 'sum', left: 1, right: 2}, console.log)
15       .act({role: 'math', cmd: 'product', left: 3, right: 4}, console.log)
16
17
18 });
```

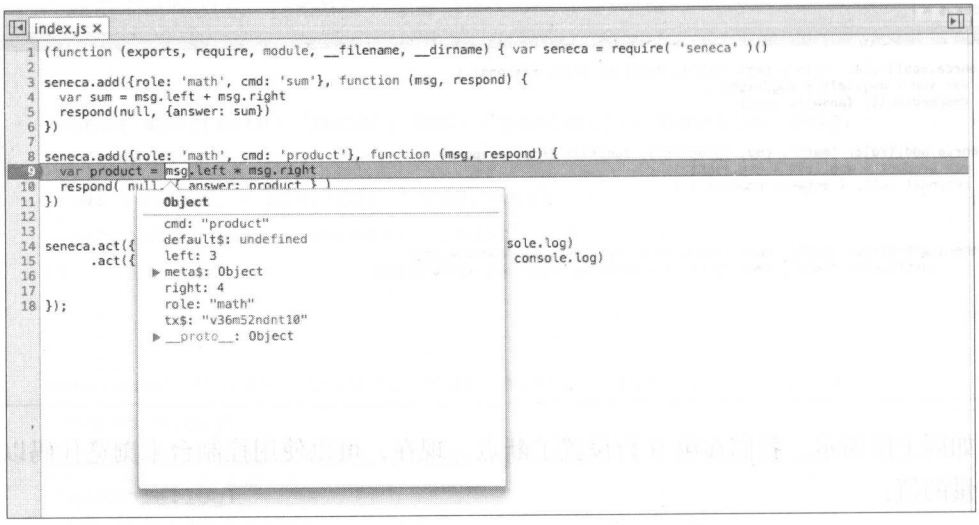
如同上图所示，我们在第 9 行设置了断点。现在，可以使用控制台来浏览代码以及获取变量的值：



如果你曾经调试过应用程序，通过顶端的操作按钮就能知道其代表的含义：

- 第一个按钮称为 play，它使应用运行到一下个断点处。
- step over 执行当前文件中的下一行代码。
- step into 将进入下一行代码，并且进一步深入调用栈，因此我们可以通过它看到调用层次。
- step out 是 step into 的反向操作。
- disable breakpoints 使得断点失效，程序在断点处不会停止。
- pause on exceptions，顾名思义，程序将在异常处停止，它对于捕捉错误非常有用。

如果单击 play 按钮，可以看见程序停在了第 9 行，如下图所示：



作为一款优秀的调试器，当光标停留在变量名上时，可以查看变量的值。

小结

本章节的内容非常紧凑。我们了解了许多能够帮助构建小型微服务生态系统的内容，如果协调使用好这些内容，它们可以很好地一起工作。有些时候，我们的方法有点过于简化，但是本书的宗旨是阐述面向微服务软件的强大之处。在现阶段，我建议读者自行围绕 Seneca 做一些测试。

网络上的文档对于学习微服务相当有帮助，并且包含了许多可以学习的例子。

Seneca 有许多用于数据存储、传输的插件，同时还有许多具有其他功能的插件，比如与用户认证相关，这些插件可以让你体验到 Seneca 的各种特性。

我们将在之后的章节中对其他一些插件进行更进一步的介绍。

5

安全性和可追溯性

如今，安全性已经成为系统中最主要的关注点之一。从大型企业中泄露的信息数量之大令人担忧，尤其是90%的信息泄露问题只需要开发工程师稍加处理就可以修复。此外，事件日志与错误追踪方面也面临着同样的问题。没有人会真正给予它们太多关注，直到我们发现无法给出审计错误时需要的日志。在本章中，我们将围绕以下几个主题来讨论如何管理安全与日志，以保证系统是安全的、可追溯的。

- **基础设施的逻辑安全：**我们将讨论如何保障软件基础设施的安全，从而在通信中提供符合行业标准的安全层。
- **应用程序安全：**我们将引入安全领域的常用技术来保证应用程序安全。一些良好的实践，例如对输出编码或者输入验证已成为行业标准，并且可以使我们避免某些灾难。
- **可追溯性：**在微服务架构中，必须能够在系统中追踪请求。这个需求可以交给 Seneca 完成，我们只需要学会如何从这个出色的框架中获取信息即可。
- **审计：**即使我们在构建软件时付出了最大的努力，也不可避免意外的发生。因此，能够重新构建调用序列并观察发生的情况是很重要的能力。我们将讨论如何使系统能够恢复审计所需的信息。

基础设施的逻辑安全

基础设施安全往往被软件工程师所忽视，因为这与他们拥有的专业领域知识截然不同。然而，现在情况发生了改变，尤其是当你的职业道路准备向运维开发工程师发展时，基础

设施安全问题不容忽视。

在本书中，我们不会过于深入地讲解基础设施安全性，而是依据拇指法则[译注1]（rules of thumb）对保障微服务安全进行介绍。

这里，我强烈建议读者学习密码学以及 SSH 的相关知识，这些是保证通信安全的主要手段。

利用 SSH 来对通信加密

任何组织机构都拥有一份严格的名单，记录着可以登录中心服务器的人员。通常，权限验证是通过用户名与密码的匹配来完成的，但是也可以使用密钥来对用户鉴权。

无论采用何种鉴权方法，通信都需要通过安全通道进行（比如 SSH）。

SSH 是 Secure Shell 的简写，它是一款用于访问远程 shell 的软件，同时它还能为访问远程服务器创建代理和隧道。

通过以下命令，我们对 SSH 是如何工作的进行说明：

```
/home/david:(develop) ✗ ssh david@192.168.0.1
The authenticity of host '192.168.0.1 (192.168.0.1)' can't be established.
RSA key fingerprint is SHA256:S22/A2/
eqxSqkS4VfR1BrcDxNX1rmfM1JkZaGhrjMbk.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.1' (RSA) to the list of known
hosts.
vagrant@192.168.0.1's password:
Last login: Mon Jan 25 02:30:21 2016 from 10.0.2.2
Welcome to your virtual machine.
```

在上述例子中，我使用 Vagrant 搭建虚拟机。Vagrant 是一款非常流行的工具，它可以自动部署环境，在 `https://www.vagrantup.com/` 上可以找到许多有用的信息。

我们在第一行执行了 `ssh david@192.168.0.1` 命令。该命令尝试以用户 `david` 的身份打开 IP 地址为 `192.168.0.1` 的主机的一个终端。

[译注1] 拇指法则，又称为经验法则，源于木工工人，他们不用尺子而是用拇指来测量、估计木材的长度，指根据经验指导行为，而不是根据完整的理论依据。

由于这是第一次对这台 IP 地址为 192.168.0.1 的机器执行该命令，我们的电脑还未信任这台远程服务器。

判断服务器是否可信任，是通过在 known_hosts 文件中维护一份受信任的名单实现的。该文件路径为/home/david/.ssh/known_hosts，具体文件路径由用户使用环境决定。

该文件中记录着一份主机列表，以及它们关联的密钥。如你所见，接下来的两行信息表明目标主机不被信任。为了验证该主机，控制台打印出了远程主机密钥的指纹信息：

```
The authenticity of host '192.168.0.1 (192.168.0.1)' can't be established.
RSA key fingerprint is
SHA256:S22/A2/ eqxSqkS4VfR1BrcDxNX1rmfM1JkZaGhrjMbk.
```

这时，用户应该对密钥进行检查，从而确认目标主机的身份。一旦确认完毕，我们可以指示 SSH 连接到服务器，获得的输出如下：

```
Warning: Permanently added '192.168.0.1' (RSA) to the list of known hosts.
```

现在，如果我们检查 known_hosts 文件，可以发现密钥信息已经被添加到受信任的主机列表中，如下所示：

```
192.168.0.1 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAxO/9E+joR8X46RL2V/wbcC15+qmQGPjfXsfpn97GV
OazzNgndR16t6WSxXmUR71fbEsjeZRYdhGp4ckkDh8AZ01MbNPuP6cKWHqy0LtOxXQR5hF/unShU8pwOPPJn8RxPB
ia3SLQ3BskfNxOrUijGqKs1JuRfeQafPuHvsOQ2kJH8PYD2UyEreHuLWiEuaiQuIguG8UiNEUkuIJEAyhD+PGVMLV
khhlTMZ+Pl0BhK7Q/9kF1e8D/ws2iBIB6I3oQx/FGW2dXuLboxODPX7iRgazf8YRv2lIWkKrh+qoD7sjTVCUgMMd4
1TbPIkNf3yrkDUQaRrfdWF0KXQ8JbNuKGhvgw==
```

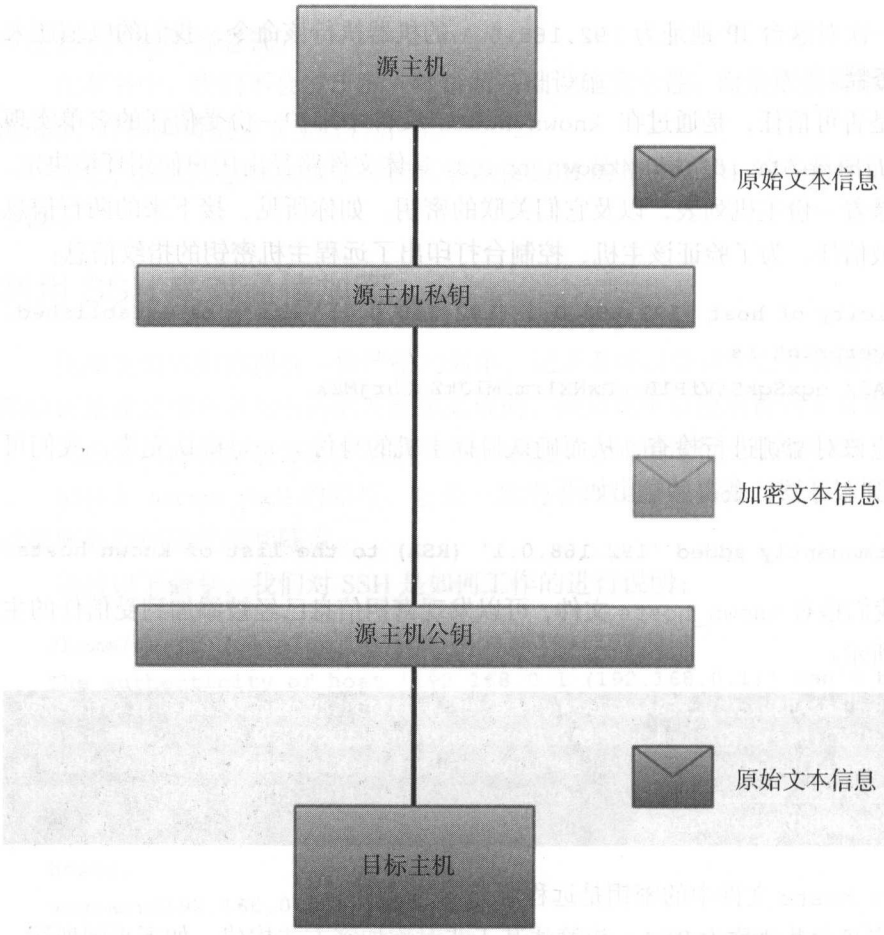
保存在 known_hosts 文件中的密钥是远程服务器的公钥。

SSH 采用的密码学算法称为 RSA，该算法基于非对称加密方式构建，如下页图所示。

这种非对称加密依赖于一组密钥：一个公钥和一个私钥。顾名思义，公钥可以共享给任何人，反之，私钥必须被私密保存。

由私钥加密过的信息只能通过对应的公钥解密，反之亦然。因此，除非有人获取到另一半密钥，否则几乎不可能拦截并解密信息。

这时，我们的计算机已经知道了该服务器的公钥，因此能够与服务器之间开始加密会话。一旦我们通过终端连接到服务器，所有的命令以及命令的结果都将被加密，并通过网络发送。



密钥还可以用于免密登录远程服务器。我们唯一需要做的就是生成本机的 SSH 密钥，并将该密钥放入远程服务器的 `authorized_keys` 文件内，该文件与 `known_hosts` 一样，位于 `.ssh` 文件夹内。

当使用微服务时，你将需要远程登录许多不同的机器，因此，该方法非常有吸引力。然而，当我们操作密钥时，需要非常小心，因为一旦某个用户泄露了密钥，我们的基础设施将面临风险。

应用程序安全

应用程序的安全性变得越来越重要。随着“云”逐渐成为大公司中基础设施的实际标

准，我们不能再依赖于单一数据中心。一直以来，每当有人开始着手新业务，其主要关注点都是如何从功能角度去构建产品。而安全性并不是主要关注点，甚至往往被忽略。

这其实是非常危险的做法，我们将帮助读者了解那些会危害到应用程序的主要安全威胁，以此来改变安全问题被忽视的现状。

应用程序开发中主要有四大类安全问题，如下所示：

- 注入
- 跨站脚本攻击
- 跨站请求伪造
- 开放重定向

到本节结束，我们将学会识别各类主要缺陷，但是我们还不能防卫恶意攻击。通常来说，一名软件工程师应该学习最新的安全技能，如同学习最新的开发技术一样。因为，无论你的产品有多好，如果它是不安全的，就会有人发现漏洞并利用它。

保持安全方面的与时俱进来应对常见威胁

正如之前提到的，安全性是应用开发中始终存在的主题。无论你开发何种类型的软件，都会存在安全隐患。

在职业生涯中，我发现不用成为全职安全工程师，也能获取最新 Web 开发领域安全知识的最好方法是：关注 OWASP 项目。OWASP 全称为 Open Web Application Security Project，他们每年都会发布一份有趣的文档，称为 OWASP Top 10。

OWASP Top 10 首次出版于 2003 年，它的目标是提高开发社区对于应用开发中常见网络安全威胁的认知。

在上一节中，我们提到软件开发者可能会遇到四大类安全问题，在接下来的几个小节中，我们将对它们进行讨论。

注入

注入是迄今为止我们可能接触到的最危险的攻击。其中，SQL 注入是最常见的影响应用的注入形式，攻击者将 SQL 代码强行植入到应用的查询中，导致应用执行了另一个完全不同的查询，其可能损害公司数据。

当然，也存在其他形式的注入，但是，我们仅关注 SQL 注入，因为在当今世界上，几乎所有应用都会使用到关系型数据库。

SQL 注入攻击由以下形式组成，即通过未经验证的来源对应用中的 SQL 查询进行注入或篡改。其中未经验证的来源可能是一个 Web 表单，或是其他任意的文本输入源。

考虑以下例子：

```
SELECT * FROM users WHERE username = 'username' AND password = 'password'
```

永远不要在数据库中直接存放原始密码，请务必对密码进行哈希或者加盐（密码安全随机值）处理，以避免彩虹表攻击。以上只是一个例子。

这条查询语句将返回指定名字与密码的用户信息。为了根据用户输入构建查询，可以参考以下代码的做法：

```
var express = require('express');
var app = express();
var mysql      = require('mysql');

var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'me',
  password : 'secret',
  database : 'test_db'
});

app.get('/login', function(req, res) {
  var username = req.param("username");
  var password = req.param("password");

  connection.connect();
  var query = "SELECT * FROM users WHERE username = '" + username
    + "' AND password = '" + password + "'";
  connection.query(query, function(err, rows, fields) {
    if (err) throw err;
```

```
    res.send(rows);
  });
  connection.end();
});

app.listen(3000, function() {
  console.log("Application running in port 3000.");
});
```

乍看之下，这是一个访问数据库 test_db 的简单程序，该程序将检查数据表中是否有匹配指定用户名与密码的用户信息，并且将该结果返回给客户端，因此，如果我们打开浏览器，浏览以下 URL 地址：http://localhost:3000/login?username=david&password=mypassword，浏览器将会呈现以下查询结果所对应的 JSON 对象：

```
SELECT * FROM users WHERE username = 'david' AND password = 'mypassword'
```

到目前为止，并没有什么问题，但是如果有人想要攻击我们的系统会发生什么？

请看以下输入：

```
http://localhost:3000/login?username=' OR 1=1 --&password=mypassword
```

如你所见，上述输入将生成如下查询语句：

```
SELECT * FROM users WHERE username = '' OR 1=1 -- AND password =
'mypassword'
```

在 SQL 中，“--”字符串用于表明其后的内容为注释，因此，有效查询语句如下：

```
SELECT * FROM users WHERE username='' OR 1=1
```

该查询将返回所有的用户信息列表，如果软件系统使用这个查询结果来判断用户是否可以登录，那么我们将面临严重的问题，这将允许甚至不知道合法用户名的人通过这种手段访问我们的系统。

这是 SQL 注入给我们带来危害的众多例子之一。

在本例中，很明显，我们把来自用户的不可信数据拼接到了查询语句中。但是，请相信，当软件系统变得更加复杂时，这个问题将不容易被发现。

避免 SQL 注入的一个方法是使用预处理语句。

输入验证

应用程序和用户主要通过表单进行交互，表单中通常包含可自由输入的文本，其可能导致系统遭受攻击。

防止有害数据进入我们服务器的最简单的方式是进行输入验证，顾名思义，输入验证就是对用户的输入信息进行合法性验证，避免上文提到的情况发生。

有两种输入验证的方式，如下：

- 白名单
- 黑名单

*黑名单*是相当危险的技术。在大多数情况下，定义不合法输入比简单定义合法输入更困难。

我们建议的方法是为用户输入的数据建立*白名单*，通过正则表达式对其进行验证，因为我们知道电话号码的格式、用户名的格式以及其他输入的格式。

然而，输入验证并不都是那么容易。如果你曾经对邮箱地址做过验证，就会明白我的意思：使用正则表达式验证邮箱地址相当复杂。

但是，实现上的困难并不能阻碍我们进行输入验证，因为，一旦省略了输入验证，可能会导致严重的安全问题。

输入验证不仅是防止 SQL 注入的“银弹”，还可以防止其他安全威胁，例如跨站脚本攻击。

在上一节的查询中，我们做了一件非常危险的事情：将用户的输入直接拼接到查询语句中。

解决该问题的方法之一就是使用转义库过滤用户输入，如下所示：

```
app.get('/login', function(req, res) {
  var username = req.param("username");
  var password = req.param("password");

  connection.connect();
  var query = "SELECT * FROM users WHERE username = '" +
    connection.escape(username) + "' AND password = '" +
    connection.escape(password) + "'";
  connection.query(query, function(err, rows, fields) {
    if (err) throw err;
```

```
    res.send(rows);
  });
  connection.end();
});
```

在本例中，mysql 库提供了一系列转义字符串的方法。

让我们看一下它是如何工作的：

```
var mysql = require('mysql');
var connection = mysql.createConnection({
  host: 'localhost',
  username: 'root',
  password: 'root'
});

console.log(connection.escape("' OR 1=1 --"))
```

以上代码对之前例子中输入的 username 参数进行了转义，得到的结果是“\' OR 1=1 --”。

如你所见，escape()方法替换了危险字符，对用户输入进行了过滤。

跨站脚本攻击

跨站脚本攻击，也称为 XSS，是主要危害 Web 应用的安全隐患。这是最常见的安全问题之一，它对于用户的潜在威胁相当大，有人利用它来盗用用户的身份信息。

这种攻击方式将代码注入到第三方网页中，然后从客户端浏览器中盗取数据。它有几种实现方法，但是目前最常见的方法是通过客户端的非转义输入。

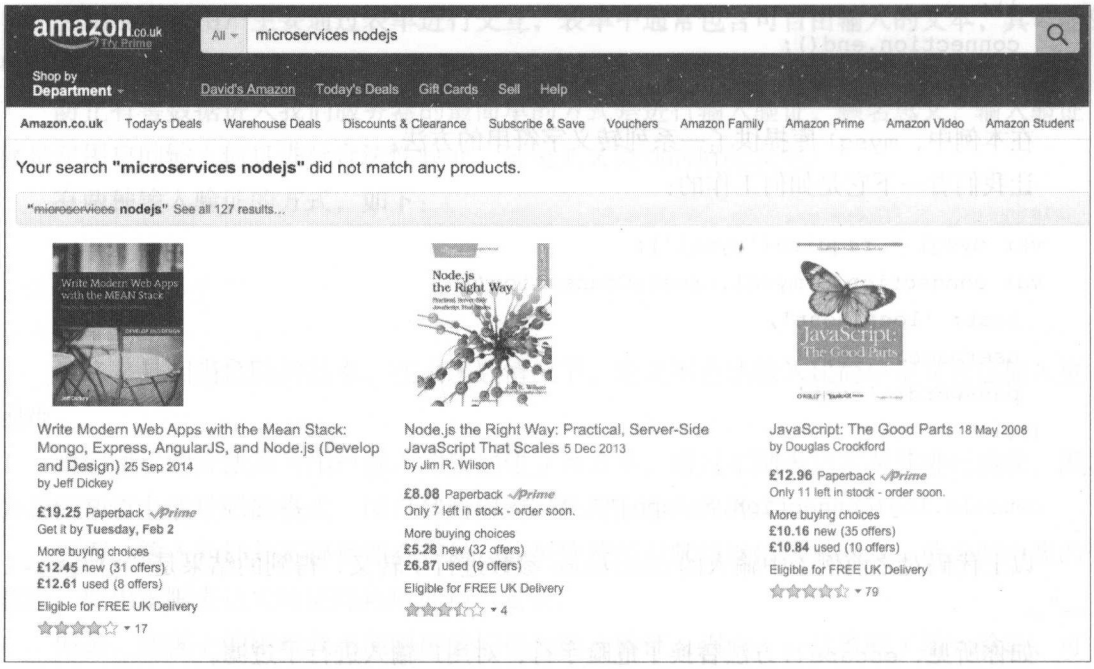
在互联网上的一些网页中，用户可以添加任意内容的评论。这些内容可能包含 script 标签，从而从远程服务器加载 JavaScript 代码，盗取会话的 cookie 或者其他有价值的信息，使得攻击者可以在远程机器上复制用户的会话。

XSS 攻击主要有两种类型：持久型与非持久型。

持久型 XSS 将特定的文本内容持久化，一旦该内容回显给网页用户时，就会触发攻击。恶意代码可以通过可随意输入的文本注入，该文本将会存入数据库，例如论坛中的评论。

非持久型的 XSS 是指通过恶意数据操作，将攻击内容插入应用的非持久化部分中。

请看以下屏幕截图：



如你所见，我们在亚马逊中搜索某本书籍，由于这本书还未出版，所以没有任何搜索结果，它提示“你搜索的‘microservices nodejs’未找到匹配商品”，这里将浏览器的输入内容拼接到输出内容中进行了回显。甚至，当我单击搜索按钮时，亚马逊重定向到了以下 URL 地址：

```
http://www.amazon.co.uk/s/ref=nb_sb_noss?url=searchalias%
3Daps&field-keywords=microservices+nodejs
```

众所周知，亚马逊是安全的。但是假如它没有防御 XSS 攻击，我们可以通过搜索关键词构建带有注入了 script 标签的请求，触发盗取会话 cookie 的攻击，这将给网站带来严重的问题。

对输出编码

防止 XSS 攻击的另一种方式就是对输出编码。在“输入验证”这一小节中，我们已经使用 `connection.escape()`进行过内容转义。同等的，我们将验证所有用户的输入信息，并对所有来自第三方的输出信息进行编码。这包括用户输入信息，以及来自外界系统的信息。

如果把范围缩小到 Web 开发领域，我们应该知道以下三种需要输出编码的格式：

- CSS
- JavaScript
- HTML

其中，问题最严重的两种是 JavaScript 和 HTML，攻击者可以轻易地利用它们盗取信息。

一般而言，无论我们选用什么框架来构建应用，该框架一定要提供对输出编码的功能。

跨站请求伪造

跨站请求伪造（CSRF）与跨站请求脚本正好相反。跨站请求脚本的问题在于，客户端信任服务端发送的数据。跨站请求伪造的问题在于，服务端信任来自客户端的数据。

跨站请求伪造是指攻击者通过 HTTP 请求将数据传送到服务器，从而盗取会话 cookie。盗取会话 cookie 之后，攻击者不仅可以获取用户信息，还可以修改该 cookie 关联的账户信息。

HTTP 对请求方法进行划分，每种方法用于指定请求将要进行的操作。其中，最常用的 4 种方法如下所示。

- `GET`：从服务器获取数据。它不会修改任何持久化数据。
- `POST`：创建服务器中的资源。
- `PUT`：修改服务器中的资源。
- `DELETE`：删除服务器中的资源。

虽然还有其他方法，例如 `PATCH`、`CONNECT` 等，但是我们只关注这 4 个方法。如你所见，这 4 个方法中有 3 个可以修改服务器中的数据。而一个持有有效会话的用户可能窃取数据、创建支付、订购商品等。

避免跨站伪造请求攻击的一种方法是使用跨站请求令牌，以防卫来自终端的 `POST`、`PUT` 和 `DELETE` 请求。

参看以下 HTML 表单：

```
<form action="/register" method="post">
  <input name="email" type="text" />
  <input name="password" type="password" />
```

```
</form>
```

该表单描述了一个能有效说明问题的场景：用户在我们的网站进行注册，虽然是一个很简单的场景，但能有效说明问题，而其中也的确存在缺陷。

从表单内容可以看出，我们仅仅指定了提交的 URL 地址以及参数列表。因此，攻击者可以使用一个小脚本不断发送 POST 请求，请求体中包含 email 和 password 参数，从而在短短几分钟内就能注册成百上千个账户。

现在，观察以下表单内容：

```
<form action="/register" method="post">
  <input name="email" type="text" />
  <input name="password" type="password" />
  <input name="csrftoken" type="hidden"
    value="as7d6fasd678f5a5sf5asf" />
</form>
```

你可以发现两个表单的区别：后者多了一个隐藏参数 csrftoken。

这个参数是每个表单生成时随机生成的字符串，我们可以将它添加到每个表单中。

一旦表单被提交，参数 csrftoken 会被验证，只允许令牌合法的请求通过，并且生成新的令牌返回给该页面。

开放重定向

有些时候，应用程序需要将用户重定向到其他特定的 URL 地址。比如，当用户没有权限访问一个私有的 URL 地址时，他们通常会被重定向到登录页面：

```
http://www.mysite.com/my-private-page
```

这将重定向到以下地址：

```
http://www.mysite.com/login?redirect=/my-private-page
```

这听起来很合理。用户被重定向到登录页面，一旦其通过身份验证，又将重新回到 /my-private-page 页面。

如果有人想要盗取用户信息，会发生什么？

请观察以下请求：

```
http://www.mysite.com/login?redirect=http://myslte.com
```

这是一个精心设计的请求，它将用户重定向到 myslte.com 而不是 mysite. com（注意是 l 而不是 i）。

某些人将 myslte.com 的界面做得和 mysite.com 登录界面一样，然后在社交媒体上散布该 URL，如果有人重定向到恶意页面并输入账户密码，他的账号将被窃取。

上述问题的解决方案非常简单：禁止用户重定向到未授信的第三方网页。

同样，这个解决方案的最佳实现方式是将可重定向的目标主机列入白名单。也就是说，我们的软件不会把用户重定向到未知的网页。

有效的代码审阅

系统而全面的代码审阅是减少应用中安全缺陷的最有效的方式之一。但是代码审阅通常最终总会成为一个各种意见和个人偏好混杂的“垃圾场”，其不仅没有提高代码质量，还可能引发一些暴露系统漏洞的临时变更。

在产品开发的生命周期中，一个慎重的安全代码审阅阶段有助于彻底减少交付生产时的 bug 数量。

软件工程师普遍存在以下问题：他们致力于构建出功能良好的产品，却没有发现产品缺陷的意识，尤其是他们自己构建的产品。所以你的代码不能只由自己进行测试（不局限于开发阶段的测试），更不用说对应用的安全测试了。

然而，我们经常进行团队协作，这促使我们彼此审阅代码，但是我们必须以有效的方式来执行它。

代码审阅需要耗费和编写软件一样多的脑力，尤其是当你审阅一份复杂代码时，绝对不要在同一个功能点上耗费两个小时以上。否则，你很可能无法察觉重大的缺陷，同时对于细节的专注度也会下降到令人担忧的程度。

在基于微服务的架构中，上述情况并不是什么大问题。因为，微服务功能实现应当尽可能精炼，使他人可以在合理的时间内审阅完。尤其是你还可以和代码作者讨论他要构建的事物。

你应当遵循以下两个代码审阅阶段。

- 快速审阅代码以获得一个蓝图：它的工作机制，它使用了哪些你不熟悉的技术，它是否完成了必需的功能等。
- 根据一个条目清单来进行审阅。

你必须事先确定清单的所有条目，它是由企业所构建的软件性质决定的。

通常，在代码审阅过程中，涉及代码安全性的条目清单内容相当多，但是，我们可以将其减少到以下几点：

- 是否对所有输入都进行了合法性检验与编码？
- 是否对包括日志在内的所有输出都进行了编码？
- 是否防御了跨站点请求伪造攻击？
- 是否对存入数据库中的所有用户身份信息都进行了加密或哈希？

如果按照以上内容检查，我们将能识别出应用中最大的安全隐患。

可追溯性

可追溯性在现代信息系统中非常重要。它对于微服务是个棘手的问题，但是在 Seneca 中已经被优雅地解决，Seneca 让我们更容易地在系统中追踪请求，从而审计错误。

日志

Seneca 在日志方面做得非常好。它有许多可配置的选项，可以获取到所有运行时信息。请看一个使用日志的小例子：

```
var seneca = require("seneca")();

seneca.add({cmd: "greeter"}, function(args, callback){
  callback(null, {message: "Hello " + args.name});
});

seneca.act({cmd: "greeter", name: "David"}, function(err, result) {
  console.log(result);
});
```

这是一个最简单的 Seneca 应用，按以下方式运行：

```
seneca  node index.js
2016-02-01T09:55:40.962Z 3rhomq69cbe0/1454579740947/84217/- INFO hello Se
neca/1.0.0/3rhomq69cbe0/1454579740947/84217/-
{ message: 'Hello David' }
```

以上是使用默认日志配置运行应用的结果。除了 `console.log()` 方法打印出的信息

外，还有一些关于 Seneca 的内部信息。有时候，你可能只需要应用的输出日志，从而可以无干扰地调试应用程序。在这种情况下，只需运行如下命令：

```
seneca  node index.js --seneca.log.quiet
{ message: 'Hello David' }
```

然而，有时候系统中存在一些不正常的行为（或者是来自所用框架的 bug），你想要获取所有相关的信息，Seneca 同样能提供支持，命令如下：

```
seneca  node index.js --seneca.log.print
```

上述指令可能会打印出无数没用的信息，为了减少 Seneca 产生的日志信息数量，可以对输出的日志进行更精细的控制，示例如下：

```
2016-02-01T10:00:07.191Z dyy9ixcavqu4/1454580006885/85010/- DEBUG
register install transport {exports:[transport/utils]} seneca-8t1dup
2016-02-01T10:00:07.305Z dyy9ixcavqu4/1454580006885/85010/- DEBUG
register init seneca-y9os9j
2016-02-01T10:00:07.305Z dyy9ixcavqu4/1454580006885/85010/- DEBUG plugin
seneca-y9os9j DEFINE {}
2016-02-01T10:00:07.330Z dyy9ixcavqu4/1454580006885/85010/-
DEBUG act root$      IN o5onzziv9i7a/b7dtf6v1u9sq cmd:greeter
{cmd:greeter,name:David} ENTRY (mnb89) - - -
```

以上几行是从前文代码示例的日志输出中随机截取的，可以看出，它们是 Seneca 框架不同 action（例如 plugin、register、act）的 debug 级别的日志。为了进行过滤，可以限定所需的日志级别和 Seneca action。示例如下：

```
node index.js --seneca.log=level:INFO
```

这将只输出 INFO 级别的日志。

```
seneca node index.js --seneca.log=level:INFO
2016-02-04T10:39:04.685Z q6wnh8qmm113/1454582344670/91823/- INFO hello
Seneca/1.0.0/q6wnh8qmm113/1454582344670/91823/-
{ message: 'Hello David' }
```

你也可以通过 Seneca 的 action 类型来过滤日志，这相当值得注意。当需要在微服务中审计一个故障时，首先需要获知业务流中发生的一系列事件，这可以通过 Seneca 提供的日志过滤方式实现，只需简单地执行以下命令：

```
node index.js --seneca.log=type:act
```

输出如下：

如你所见，所有输出行都和 act 类型是相关的。如果我们从上往下追踪命令输出，就能确切地获得 Seneca 响应的一系列事件以及它们的顺序。

请求追踪

请求追踪是一种非常重要的行为。如果你在金融界工作，它甚至可以成为一种法律上的需求。同样的，Seneca 在请求追踪上也非常出色。对于每个调用，Seneca 都会产生一个唯一标识符，这个标识符可以在所有调用路径上被追踪到。如下所示：

```
var seneca = require("seneca")();

seneca.add({cmd: "greeter"}, function(args, callback){
  console.log(this.fixedargs['tx$']);
  callback(null, {message: "Hello " + args.name});
});
seneca.act({cmd: "greeter", name: "David"}, function(err, result) {
  console.log(this.fixedargs['tx$']);
});
```

我们在日志中打印 Seneca 生成的标识 ID，并将日志输出到终端，执行上述代码，将获得如下输出：

```
2016-02-04T10:58:07.570Z zl0u7hj3hbeg/1454583487555/95159/- INFO hello
Seneca/1.0.0/zl0u7hj3hbeg/1454583487555/95159/-
3jlroj2n91da
3jlroj2n91da
```

你可以看到 Seneca 如何追踪每个请求：框架给每个请求分配了一个 ID 并且在服务端点间传播。在本例中，所有的服务端点都在同一台服务器上，但是即使我们在不同机器上

进行分布式部署，ID 也能保持不变。

通过唯一的 ID 标识符，我们可以在系统中重建用户的数据流，通过关联的时间戳对请求进行排序，同时，还能精确地展示出某个用户的行为、每个操作的耗时、延时相关的潜在问题等。通常来说，将日志和“断路器”的输出信息相结合，可以让工程师花费更少时间解决问题。

审计

到目前为止，我们一直通过 `console.log()` 来完成日志输出。这是一个糟糕的做法，因为它破坏了日志的格式，并且把内容打印到了标准输出上。

Seneca 同样解决了这个问题：

```
var seneca = require("seneca")();

seneca.add({cmd: "greeter"}, function(args, callback){
  this.log.warn(this.fixedargs['tx$']);
  callback(null, {message: "Hello " + args.name});
});

seneca.act({cmd: "greeter", name: "David"}, function(err, result) {
  this.log.warn(this.fixedargs['tx$']);
});
```

输出如下：

```
seneca  node index.js
2016-02-04T11:17:28.772Z wo10oa299tub/1454584648758/98550/- INFO hello
Seneca/1.0.0/wo10oa299tub/1454584648758/98550/-
2016-02-04T11:17:29.156Z wo10oa299tub/1454584648758/98550/- WARN - - ACT
02jlpyiux70s/9ca086d19x7n cmd:greeter 9ca086d19x7n
2016-02-04T11:17:29.157Z wo10oa299tub/1454584648758/98550/- WARN - - ACT
02jlpyiux70s/9ca086d19x7n cmd:greeter 9ca086d19x7n
```

如上所示，我们使用日志器输出标识符 ID，用 WARN 级别日志信息代替简单的控制台输出。从现在开始，为了获取真正需要的信息，可以使用 Seneca 的日志过滤器来隐藏其他 action 输出。

Seneca 提供了以下 5 种级别的日志。

- DEBUG：用于开发时的程序调试和生产系统中的问题跟踪。
- INFO：标记重要的事件信息，例如一个事务的开始和结束。
- WARN：告警级别，表明系统发生了不正常的行为，但并不严重。用户往往不会受到影响，指示某些事情不太正常。
- ERROR：用于记录错误。通常来说，用户会因为这些错误受到影响，操作流程终止。
- FATAL：这是最具灾难性的级别，通常在不可恢复的错误发生及系统功能不正常时使用。

通过相关方法可以产生不同级别的日志，如上文所述，我们调用 `this.log.warn()`输出告警日志，通过 `this.log.fatal()`输出致命错误日志，其他级别日志同理。

请把调整应用中的日志作为开发过程的一部分，否则当生产环境发生问题时，你会后悔缺少排查问题的信息。

通常，INFO、DEBUG、WARN 是最常用的三种日志级别。

HTTP 状态码

HTTP 状态码经常被忽视，但它们是对远程服务器响应进行标准化的重要机制。

当一个程序或者用户发送一个请求到某个服务器时，可能发生以下情况：

- 请求成功
- 验证失败
- 产生服务器错误

如你所见，可能性是无穷尽的。现在，存在一个问题：服务器间用 HTTP 进行通信，那么获取到 HTTP 状态码后，服务器该如何处理？

HTTP 协议用非常优雅的方式解决了这个问题：每个请求必须包含一个 HTTP 状态码，这些状态码的值在一定的范围内，标示状态码的性质。

1xx，消息性的

范围在 100 ~ 199 的状态码是纯消息性的，其中最值得注意的是状态码 102，它表示一

个操作正在后台发生，可能需要一段时间完成。

2xx，成功状态码

成功类型的状态码用于表示 HTTP 请求成功，这是最通用也是最被期望的状态码。在这个范围内，最通用的状态码如下所示。

- `200: Success`：表示完全成功，远端没有错误。
- `201: Created`：主要用于 REST API，表示用户请求已经在远程服务器上创建了新的资源。
- `203: Non-authoritative information`：原始状态码为 200，但是当请求通过了一个转换代理时使用该状态码。
- `204: No Content`：虽然表示成功，但是服务器没有内容返回。有时候，即使没有内容返回，API 也会返回 200。
- `206: Partial Content`：用于分页响应，请求头指定了客户端可接受的数据范围和偏移，如果返回结果比这个范围大，服务器将返回 206，表明还有剩余的数据需要继续返回。

3xx，重定向

范围在 300 ~ 399 的状态码表明客户端必须采取额外的操作来完成请求。在这个范围中，最通用的状态码如下所示。

- `301: Moved permanently`：表明客户端请求的资源已经被永久地移动到另一个地方。
- `302: Found`：表明用户因为某种原因需要进行临时重定向，但是一些浏览器的早期实现都将这种状态码当作 `303（See Other）`处理，最终导致又引入了 303 和 `307（Temporary Redirect）`两种状态码来明确区分重叠的行为。
- `308 : Permanent Redirect`：顾名思义，用于表示一个资源的永久重定向。它可能与 301 产生混淆，但还是存在微小的差异，状态码 308 不允许修改 HTTP 的方法。

4xx，客户端错误

范围在 400 ~ 499 内的状态码表示错误由客户端产生，即请求存在问题。这个范围非常重要，因为 HTTP 服务端用它代表客户端请求存在何种错误。

在这个范围中，最通用的状态码如下所示。

- `400 Bad Request`：用户请求语法错误，可能是参数缺失或者某些值验证失败。
- `401 Unauthorized`：缺少用户权限验证，通常，一个有效登录将解决这个问题。
- `403 Forbidden`：类似 401，但是这种情况表明用户没有足够的权限。
- `404 Not Found`：服务端没有找到资源，当你跳转到一个不存在页面时，将得到这个错误。

5xx，服务器端错误

这个范围表示服务端存在运行错误，当返回一个 5xx 状态码时，表示服务端发生了一个无法通过客户端修复的错误。

在这个范围中，最常用的状态码如下所示。

- `500 Internal Server Error`：表示错误发生在服务端的软件内，没有更多的信息可以透露。
- `501 Not Implemented`：客户端发送了一个还不支持的请求。
- `503 Service unavailable`：服务端因为某个原因不可用，可能是资源过载或者服务器停机。

为什么 HTTP 状态码对于微服务如此重要

“不要重复发明轮子”是我在构建软件时最喜欢的准则之一。HTTP 状态码已然成为一项标准，所有人都能明白不同状态码表示的结果。

当构建微服务时，请记住你的系统会和代理、缓存或其他服务交互，它们通过 HTTP 进行通信，从而根据服务端的响应执行操作。

断路器模式就是最佳的例子，无论你如何实现它，使用什么软件，一个断路器必须知道“状态码为 500 的请求表示错误”，从而产生相应的“断路”。

总而言之，保持应用的状态码尽量精确是一个很好的习惯，因为这将给你的系统带来长远的好处。

小结

在本章中，你已经学会了如何构建安全的软件（不仅限于微服务范畴内），虽然这是个

可以叙述一整本书的话题。安全方面目前的问题是：企业通常将这类投资视为“烧钱”，但是它们的这种看法不正确。我是一个 80-20 原则的狂热粉丝：20%的时间将创造产品 80%的功能点，剩余 20%的功能点将需要占用 80%的时间。

在安全方面，我们应当以 100%的覆盖率为目标，本章 80%的内容涵盖了大部分安全相关的用例。无论如何，如我之前所述，一个软件工程师应该在安全方面与时俱进，因为应用系统中的一个安全缺陷将会是摧毁一家企业最简单的方式。

此外，我们还讨论了可追溯性和日志。它们在现代软件工程中最容易被忽视，但却越来越重要，尤其是当你以微服务方式来构建软件时。

6

Node.js 微服务的测试及文档化

到目前为止，我们已经完成了微服务的开发，并对围绕软件组件构建过程中的一些框架进行了讨论，现在是时候来对这些服务进行测试了。测试是对所建软件进行验证的一项活动，而验证的范畴通常很广。在本章中，我们将学习如何对微服务进行测试。该测试并不仅仅停留在功能性的视角，同时还涉及对应用性能的测试以及不同组件之间的集成等其他方面。我们会使用 Node.js 来构建代理，并通过该代理来对服务的输入输出进行检查，从而验证实际的实现效果是否符合起初的设计，进而为我们采用 JavaScript 来快速搭建原型注入信心。

如今，通过 A/B 测试来发布功能已然成为一种趋势，通过 A/B 测试，可以仅对某些类型的用户启用新功能，随后可以收集各种指标来观察系统表现有什么变化。在本章中，我们将创建一个能以可控方式推送新功能的微服务。

另一方面，我们还将对应用进行文档化。然而不幸的是，文档化往往是传统软件开发中被遗忘的一环：我从未发现有某家公司的文档能 100%覆盖新开发者所需的所有信息。

我们将会在本章覆盖以下话题。

- **功能性测试：**在本章中，我们将会学习如何测试微服务以及优秀的测试策略是怎样的。除此之外，我们还将介绍一款用于对 API 进行人工测试的工具 Postman，同时还将采用 Node.js 来构建可用于监控连接的代理。
- **微服务的文档化：**我们将学习如何采用 Swagger 并使用开放 API 标准来对微服务进行文档化。同时还将采用一款开源工具尝试从 YAML 定义生成代码。

功能性测试

测试在软件构建过程中通常是一件耗时且并没有得到足够重视的事情。

想想一家公司是如何发展的：

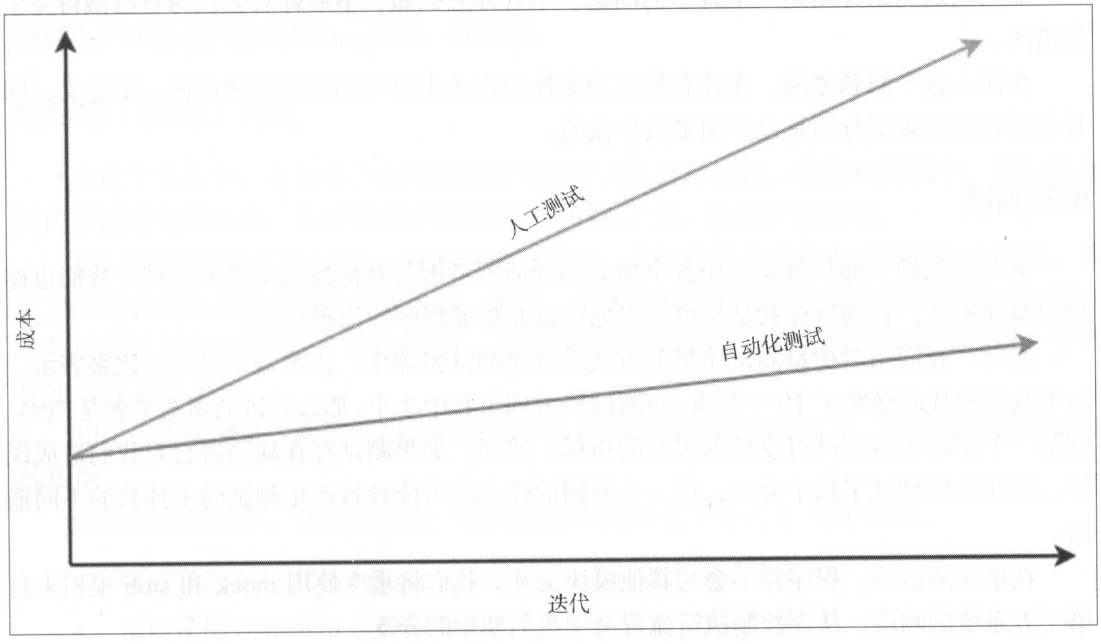
1. 某个人有了一个好点子。
2. 一个工程师或产品经理共同将系统构建出来。
3. 公司将产品推向市场。

除了最基本的人工测试，我们几乎没有更多的时间来进行其他测试。如果你去网上查阅一下相关资料就会发现，通常正确的测试需要花费 40%的开发时间，这一数据再一次颠覆了我们对测试时间占比的认识。

自动化是一种解决效率问题的好办法。而单元测试、集成测试以及端到端的测试都是一种自动化的形式。通过让计算机来测试软件，可以大大降低我们花费在软件验证上的精力。

想想软件开发是怎样进行的。即便公司乐于声称自己是敏捷的，然而事实上也只做到了每个软件项目都拥有一定程度的迭代开发，并且每个周期中都拥有测试环节。但是通常来说，我们忽视了其在新功能交付中的重要性。

如下图所示，通过将大部分（或大量）测试自动化，我们为公司节省了成本。



如果测试做得正确将会为我们节省成本，这里的关键是要做得正确，而要做到这一点往往并不容易。多少测试才是足够多了呢？我们的测试需要覆盖应用的每个角落吗？我们真的需要深度的性能测试吗？

人们对于这些问题往往有着不同的观点，有意思的是并没有放之四海皆准的答案，因为一切取决于你系统的特性。

在本章中，我们将广泛学习各种测试技术。这并不意味着我们需要在测试计划中用上所有这些技术，但是至少你要在以后能有意识地去使用这些测试技术。

在过去的七年里，Ruby on Rails 在一定程度上创建了一个通往新范式的大趋势，该新的范式就是测试驱动开发（TDD）。如今，大部分新的开发平台都是以 TDD 的思想来构建的。

就我个人而言，并不算是 TDD 的狂热分子，但是我喜欢选用其中好的部分。在开发之前做好测试计划将有助于创建出高内聚的模块，并定义出清晰且易于测试的接口。在本章中，我们并不会深入探讨 TDD，但是会多次提及 TDD 并解释如何在一个 TDD 测试计划中应用好提到的测试技术。

自动化测试的金字塔

如何制定测试计划是一个棘手的问题。不管你怎么做，事后你总会认为自己做得完全是错的。

在深入这个问题之前，先让我们从功能性的视角来对不同类型的测试做一下定义，同时来看看这些测试分别是设计用来做什么的。

单元测试

*单元测试*是一种只覆盖应用各个独立部分而不考虑与其他模块集成的测试。我们也称它为*白盒测试*，它的目标就是尽可能多地覆盖并验证程序的代码分支。

通常，我们用以衡量测试质量的方式是观察测试覆盖率，该覆盖率以百分比来表示。如果我们的代码横跨了 10 个分支，而测试覆盖到了其中 7 个，那么测试的覆盖率就是 70%。这是一个用以衡量测试可靠性的很好的指标。然而，如果测试存在缺陷将会对我们造成误导。即使我们测试了每个分支，但一个不同的输入将可能导致产生测试尚未捕获的不同输出。

在单元测试中，程序并不会与其他模块交互。我们将重度使用 mock 和 stub 模拟来自第三方系统的响应，从而控制执行流程命中我们期望的分支。

集成测试

正如字面所表明的，集成测试是设计用来验证应用环境中各模块之间集成的。它们并不是用来验证代码分支的，它们验证的是业务单元，在该测试中，我们将数据存入数据库，或者调用第三方的 Web 服务，又或者是调用架构中的其他微服务。

这些测试是检验服务是否按预期方式执行的绝佳工具，但有的时候，这些测试并不容易维护（多半时候）。

据我多年的经验来看，很少有公司能将集成测试做对，而之所以造成这样的情况的原因有如下几个：

- 一些公司认为集成测试成本过高（然而这也是事实），因为它需要额外的资源（例如数据库和额外的机器）。
- 其他一些公司尝试通过单元测试来覆盖所有的业务用例，这就完全依赖于业务用例的设计是完善的。但事实上，单元测试中的假设（mock）过于理想化，这样反而给我们带来了虚假的信心。
- 有的时候，集成测试会被当作单元测试来验证代码的分支，这样你就需要创造环境来让集成测试命中目标分支，这会相当耗时。

无论你认为自己有多聪明，集成测试必定是你想要做对的一件事，因为它是在软件发布到生产环境前拦截集成 bug 的第一道屏障。

端到端（E2E）测试

在这个环节中，我们将会对应用的真实运行方式进行验证。在集成测试中，我们是在代码层面调用服务的。这意味着需要构建服务的上下文，然后发起调用。

它与端到端测试的区别是：在端到端测试中，我们需要实实在在地部署整个应用，然后只发起必要的调用来执行目标代码。然而，很多时候工程师可以决定将两种类型的测试（集成测试与端到端测试）结合使用。在很多流行的框架中，我们可以像集成测试一样来快速运行 E2E 测试。

相比于集成测试，端到端测试的目标并不是测试应用的所有分支，而只是测试设计好的用例。

在端到端测试中，我们能发现一些测试的不同形态（范式），如下所示：

- 可以通过发起 JSON 请求来测试 API（或者其他类型的请求）。
- 可以使用 Selenium 来模拟 DOM 单击，从而对 UI 进行测试。

- 可以使用一种叫作行为驱动开发（BDD）测试的新范式，该范式将用例映射到应用的行为中（形式诸如单击 UI、请求 API 等），从而执行该应用的用例。

端到端测试通常比较脆弱，且很容易失效。根据应用的实际情况，应该适当放宽这些测试，因为它的性价比相当低。但是无论如何，我还是建议要进行端到端测试，且至少覆盖到最基本和最重要的业务流程。

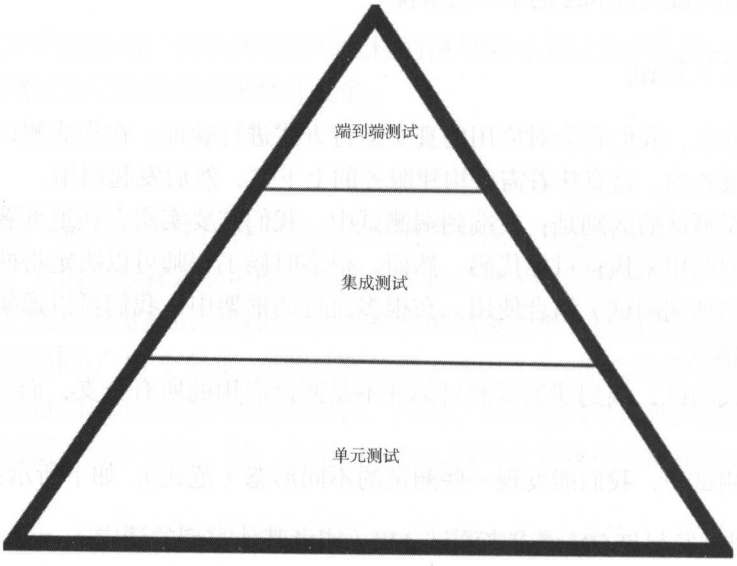
多少测试才足够

像下面这样的问题并不好回答，尤其是在业务节奏快的初创公司：

- 我们是否拥有了足够多的集成测试？
- 我们是否应该以 100%的单元测试覆盖率为目标？
- 既然 Selenium 测试每两天就会无缘无故失效，我们为何还要使用它？

在测试覆盖率与耗时之间通常是需要进行平衡的。对于这些问题来说，也没有简单且唯一的答案。

这些年来，我发现的唯一有借鉴意义的是一个叫作测试金字塔的概念，如下图所示。请稍作思考，回顾一下你之前参与的项目，你总共拥有过多少测试？其中集成测试和单元测试的比例又分别是多少？端到端测试呢？



测试金字塔

上图的金字塔展示了这些问题的答案。在一个健康的测试计划中，我们应该拥有大量单元测试、适量的集成测试以及少量的 E2E 测试。

这样做的原因很简单，单元测试可以捕获到大部分问题。通过命中代码的不同分支，几乎可以对应用中每个功能性用例进行验证，以至于在我们的测试计划中单元测试占了最大的比重。根据我的经验，在一个平衡的测试计划中，差不多有 70%的测试是单元测试。然而，在面向微服务的架构中，尤其是采用了像 Node.js 这样的动态语言的情况下，该图的结果只要向下稍作调整，对我们的测试来说还是有效的。背后的原因主要是因为 Node.js 允许我们快速地编写集成测试，以至于可以采用集成测试来替代部分单元测试。

测试是一项理论依据充分且错综复杂的流程。想要尝试超越从而偏离当前的方法论可能会导致创建出难以维护或难以信任的测试套件。

集成测试负责捕获集成问题，如下所示：

- 我们的代码可以成功调用 SMS 网关吗？
- 是否可以正常连接到数据库？
- 这个 HTTP 头部是由我们的服务发送的吗？

根据我的经验，差不多有 20%的测试是集成测试；其中大部分关注正常的执行流程，以及一部分依赖于第三方组件的异常流程。

当进行到 E2E 测试的时候，应该有效控制 E2E 测试的量，建议仅测试应用的主流程，而不要深入到过于细节的部分。这些细节的问题应该已经在单元测试和集成测试阶段被捕获到，并可以轻松地通过失败事件来修复。然而，这里还有一点需要注意：当采用 Node.js 来测试微服务时，大多数时候（90%）集成测试与端到端测试是同一回事。根据 Node.js 的动态特性，我们可以从集成测试的视角（即整个服务器都运行起来）来测试 rest API，但是事实上，同样也会测试与其他模块集成时的代码行为。我们将在这章的后续部分看到一个例子。

采用 Node.js 测试微服务

Node.js 是一门令人印象深刻的语言。大量围绕开发工作各个方面的代码库叫人惊艳不已。不管你想用 Node.js 来完成怎样高难度的任务，它都会有相对应的 npm 模块供你使用。

在与测试相关的方面，Node.js 拥有非常强大的代码库集合，其中有两个尤其流行：Mocha 和 Chai。

它们几乎是应用测试方面的行业标准，并且很容易维护和升级。

另一个有趣的库叫 Sinon.JS，它可以用于 mock、spy 及生成方法 stub。我们将会在后续的几个小节中讨论这几个概念，这个库主要用于模拟与第三方的集成，从而避免我们在测试时真正与第三方交互。

Chai

该库是一个 BDD/TDD 断言库，可用于协同其他任何库创建高质量的测试。

断言是一个代码语句，它要么全部被满足，要么就抛出错误并停止测试，使测试主动失败：

```
5 should be equal to A
```

在该语句中，如果变量 A 的值是 5，那么该语句将会正确返回。这是一个强大的工具，可以编写出让人非常容易理解的测试代码。在使用 Chai 的时候，我们可以通过以下三个不同的接口来访问断言：

- should
- expect
- assert

其实在每天结束的时候，我们可以只使用一种接口来对每一个条件进行检查。而这个库之所以提供给我们如此丰富的接口是为了让我们能利用它们编写出更加清晰、简单和易于维护的测试代码。

让我们来安装这个库：

```
npm install chai
```

安装完毕后会产生如下输出：

```
├── assertion-error@1.0.1
├── type-detect@1.0.0
└── deep-eql@0.1.3 (type-detect@0.1.1)
```

这意味着 Chai 依赖于 assertion-error、type-detect 和 deep-eql。正如你所看到

的，这是一个良好的开端，现在我们可以采用简单的指令来对复杂的语句进行检查，检查项包括深度相等或类型匹配。

像 Chai 这样的测试库并不是我们应用的直接依赖，它只是一个开发阶段的依赖。我们需要用它们来完成应用开发，但是不应该把它们交付到生产环境中去。因此，需要重构 `package.json`，将 Chai 添加到 `devDependencies` 的依赖标记中去，如下所示：

```
{
  "name": "chai-test",
  "version": "1.0.0",
  "description": "A test script",
  "main": "chai.js",
  "dependencies": {
  },
  "devDependencies": {
    "chai": "*"
  },
  "author": "David Gonzalez",
  "license": "ISC"
}
```

这样一来，像 Chai 这样的开发库就不会被交付到软件的生产环境中去了，而应用本身无须进行任何修改。

一旦安装完 Chai，我们就可以开始操作这些接口了。

BDD 风格的接口

Chai 拥有两种不同风格的 BDD 接口。选择哪种风格是需要斟酌的，我个人的建议是选择在相应场景中让你觉得更对胃口的那种接口。

让我们从 `should` 接口开始。这是其中一种 BDD 风格的接口，使用形式类似于自然语言，我们可以创建一个断言来看看测试是否成功：

```
myVar.should.be.a('string')
```

为了能够编写如上的语句，我们需要在程序中导入 `should` 模块：

```
var chai = require('chai');

chai.should();
```

```
var foo = "Hello world";
console.log(foo);

foo.should.equal('Hello world');
```

虽然它看上去有点像黑魔法，但是由于它使用起来与自然语言非常相似，所以当我们使用它来验证代码是否符合条件时显得格外方便：*foo 应该等于'Hello world'*，这简直是对我们测试代码的直译。

Chai 提供的第二种 BDD 风格的接口是 `expect`。虽然它与 `should` 非常相似，但是语义上略有差别，它表示为结果设置必须要符合的预期。

让我们来看看下面的例子：

```
var expect = require('chai').expect;

var foo = "Hello world";

expect(foo).to.equal("Hello world");
```

正如你所看的，这种风格很眼熟：这是一个用以检查测试条件是否成功符合预期的连贯接口，如果没有符合预期又会怎样呢?

让我们来执行一个简单的 Node.js 程序，其中的一个测试条件会失败：

```
var expect = require('chai').expect;
var animals = ['cat', 'dog', 'parrot'];
expect(animals).to.have.length(4);
```

现在让我们来执行一下上面的脚本，前提是你已经安装了 Chai：

```
code/node_modules/chai/lib/chai/assertion.js:107
      throw new AssertionError(msg, {
            ^

AssertionError: expected [ 'cat', 'dog', 'parrot' ] to have a length of 4
but got 3
      at Object.<anonymous> (/Users/dgonzalez/Dropbox/Microservices with
Node/Writing Bundle/Chapter 6/code/chai.js:24:25)
      at Module._compile (module.js:460:26)
```

```
at Object.Module._extensions..js (module.js:478:10)
at Module.load (module.js:355:32)
at Function.Module._load (module.js:310:12)
at Function.Module.runMain (module.js:501:10)
at startup (node.js:129:16)
at node.js:814:3
```

测试失败并抛出了异常。如果所有的条件都通过校验，且没有异常抛出来，那么测试将会成功。

正如你所见，我们可以在测试中使用一些自然语言中的词汇，包括 `expect` 和 `should` 接口。你可以在 Chai 的文档中找到完整的词汇列表（`http://chaijs.com/api/bdd/#-include-value-`），让我们从列表中挑几个有趣的来看看。

- `not`：该词用于对后续链中的断言进行取反。举个例子，`expect("some string").to.not.equal("Other String")`将会通过测试。
- `deep`：该词当属列表中最有趣的词之一。它可用于对象之间的深比较，这是一种最快捷的全等比较方式。举个例子，如果 JavaScript 对象 `foo` 拥有一个叫作 `name` 的属性，且其对应的值等于`"David"`，那 `expect(foo).to.deep.equal({name: "David"})`就会成功。
- `any/all`：该词用于检查字典或对象是否含有给定列表中的键，所以如果 `foo` 包含键 `"name"` 或 `"surname"` 的话，`expect(foo).to.have.any. keys("name", "surname")`将会成功，而只有当 `foo` 同时包含键`"name"`和`"surname"`时，`expect(foo).to.have.all.keys("name", "surname")`才会成功。
- `ok`：该词也很有趣。你应该也知道，JavaScript 拥有很多陷阱，比如表达式的 true/false 求值。通过采用 `ok`，我们将无须关心其中混乱的细节，只需采用类似如下的表达式即可：
  - `expect('everything').to.be.ok`：`'everything'`是一个字符串，所以对它的求值将会成功。
  - `expect(undefined).to.not.be.ok`：Undefined 在 JavaScript 世界中并不是一个正常的值，所以断言将成功。
- `above`：该词非常有用，可以检查数组或集合中是否包含指定阈值以上的元素个数，比如：`expect([1,2,3]).to.have.length.above(2)`。

正如你所见，Chai API 的连贯断言接口是很丰富的，可以让我们编写出描述性较强的测试，且更容易维护。

现在，你可以问问自己，为什么需要两种不同风格且干着类似工作的相似接口呢？好吧，它们在功能上是差不多的，那么来进一步看看细节之处的差别：

- `expect` 为链式语言提供了起始点。
- `should` 则扩展了 `Object.prototype` 的签名，并为 JavaScript 中的每个对象添加了对链式语言的支持。

从 Node.js 的角度来看，这两种风格的接口都是没问题的。然而 `should` 方式改造了 `Object` 的原型，这可能是在选择使用它前需要考虑的一个显得略为偏执的因素，因为这样做是带有侵入性的。

断言接口

断言接口对应了最常见的传统测试断言库。在这种风格的接口中，我们需要指定具体的测试内容，同时无法再使用像之前连贯的链式表达式：

```
var assert = require('chai').assert;
var myStringVar = 'Here is my string';
// 不带message参数
assert.typeOf(myStringVar, 'string');
// 带message参数
assert.typeOf(myStringVar, 'string', 'myStringVar is not string type.');
// 对长度进行断言
assert.lengthOf(myStringVar, 17);
```

其实只要你曾使用过现有的任何语言的测试库，我都没有必要在这方面做过多的介绍。

Mocha

在我看来，Mocha 是我职业生涯中使用过的最方便的测试框架之一。它遵循行为驱动开发测试（BDDT）的原则，每个测试都描述了应用的一个用例，并通过使用其他库的断言来验证执行代码的结果。

虽然这听上去有点复杂，但是这样做可以方便地确保代码能够覆盖到功能与技术两个视角。而我们可以将所建应用的需求反映在自动化测试中，从而验证这些需求。

让我们先从一个简单的例子开始。Mocha 与其他的任何测试库都有所不同，它定义了

自己的领域特定语言（DSL），我们需要执行这套 DSL 而不是 Node.js。该 DSL 可以认为是对 JS 语言的一套扩展。

首先我们需要在系统中安装 Mocha：

```
npm install mocha -g
```

这行命令将会产生类似如下的输出：

```
/usr/local/bin/mocha -> /usr/local/lib/node_modules/mocha/bin/mocha
/usr/local/bin/_mocha -> /usr/local/lib/node_modules/mocha/bin/_mocha
mocha@2.4.5 /usr/local/lib/node_modules/mocha
├── escape-string-regexp@1.0.2
├── supports-color@1.2.0
├── growl@1.8.1
├── diff@1.4.0
├── commander@2.3.0
├── jade@0.26.3 (commander@0.6.1, mkdirp@0.3.0)
├── debug@2.2.0 (ms@0.7.1)
├── mkdirp@0.5.1 (minimist@0.0.8)
└── glob@3.2.3 (graceful-fs@2.0.3, inherits@2.0.1, minimatch@0.2.14)
```

现在，我们的系统中已经多了一个可以使用的命令：mocha。

下一步使用 Mocha 来编写一个测试：

```
function rollDice() {
  return Math.floor(Math.random() * 6) + 1;
}

require('chai').should();
var expect = require('chai').expect;

describe('When a customer rolls a dice', function(){

  it('should return an integer number', function() {
    expect(rollDice()).to.be.an('number');
  });

  it('should get a number below 7', function(){
    rollDice().should.be.below(7);
  });
```

```
  it('should get a number bigger than 0', function(){
    rollDice().should.be.above(0);
  });

  it('should not be null', function() {
    expect(rollDice()).to.not.be.null;
  });

  it('should not be undefined', function() {
    expect(rollDice()).to.not.be.undefined;
  });
});
```

前面的例子很简单。这是一个具有掷骰子功能的函数，可以返回一个 1 到 6 之间的整数。现在我们需要思考一下与之相关的用例和需求：

- 数字必须是一个整数
- 数字必须小于 7
- 数字必须大于 0，点数是没有负数的
- 该函数不能返回 `null`
- 该函数不能返回 `undefined`

这已经覆盖了使用 Node.js 实现掷骰子功能的所有细枝末节的场景。以上所描述的情形都是我们确实想要测试的场景，这样做是为了后续能够放心地对软件进行改造而不会破坏现有的功能。

以上的 5 个用例准确地对应了之前我们所编写的测试。

- **我们描述了相应的场景：**用户掷骰子。
- **条件得到了验证：**函数调用后将返回一个整数。

让我们来运行一下该测试，并检查一下结果：

```
mocha tests.js
```

运行结果大致如下图所示：

```
When a customer rolls a dice
  ✓ should return an integer number
  ✓ should get a number below 7
  ✓ should get a number bigger than 0
  ✓ should not be null
  ✓ should not be undefined

5 passing (9ms)
```

正如你所看到的，Mocha 返回了一份测试结果的综合报告。在本次运行中，所有的用例都通过了，因此没有什么问题需要我们担心了。

让我们刻意来构造一个失败的测试：

```
function rollDice() {
  return -1 * Math.floor(Math.random() * 6) + 1;
}

require('chai').should();
var expect = require('chai').expect;

describe('When a customer rolls a dice', function(){

  it('should return an integer number', function() {
    expect(rollDice()).to.be.an('number');
  });

  it('should get a number below 7', function(){
    rollDice().should.be.below(7);
  });

  it('should get a number bigger than 0', function(){
    rollDice().should.be.above(0);
  });

  it('should not be null', function() {
    expect(rollDice()).to.not.be.null;
  });
```

```
  it('should not be undefined', function() {
    expect(rollDice()).to.not.be.undefined;
  });
});
```

偶尔，也会有人不小心改坏了 `rollDice()` 函数中的一段代码，从而导致函数返回了无法满足需求的数字。让我们再次运行 Mocha，如下图所示：

```
When a customer rolls a dice
  ✓ should return an integer number
  ✓ should get a number below 7
  1) should get a number bigger than 0
  ✓ should not be null
  ✓ should not be undefined


4 passing (10ms)
1 failing

1) When a customer rolls a dice should get a number bigger than 0:
   AssertionError: expected -4 to be above 0
```

现在，可以看到报告中返回了一个错误：方法调用后返回了-4，而我们原本期望的是返回一个大于 0 的数字。

另外，使用 Mocha 和 Chai 来完成 Node.js 中的这类测试提高了我们的时间效率。测试运行非常快，从而让我们能够在代码被破坏时快速得到反馈。上述测试集运行时间不到 10ms。

Sinon.JS——一个 mocking 框架

前面两节主要专注于对函数返回值的断言条件，但是如何判断那些不返回值的函数的执行情况呢？唯一正确的度量方式是检查方法是否得到了调用。同样的，如果一个模块需要调用第三方的 Web 服务，而我们并不希望在测试中真正访问远程服务器又该怎么办呢？

为了解答上述问题，我将介绍两个概念上的工具，它们是 mock 和 spy，而 Node.js 正拥有一个实现了这些工具的完美测试库：Sinon.JS。

首先来安装它：

```
npm install sinon
```

上述命令会产生如下输出：

```
sinon@1.17.3 node_modules/sinon
├── lolex@1.3.2
├── samsam@1.1.2
├── formatio@1.1.1
└── util@0.10.3 (inherits@2.0.1)
```

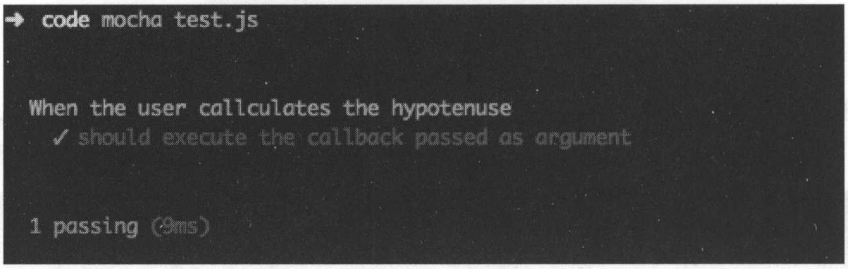
现在让我们通过一个例子来说明测试框架是怎么工作的：

```
function calculateHypotenuse(x, y, callback) {
  callback(null, Math.sqrt(x*x + y*x));
}

calculateHypotenuse(3, 3, function(err, result){
  console.log(result);
});
```

这段简单的脚本可以用于计算一个三角形的斜边长度，它接收两条直角边的长度作为其输入。其中一个测试是要验证回调函数 callback 是否会以我们提供的参数列表进行执行。我们只需通过 Sinon.JS 调用一次 spy 就可以完成这样的工作：

```
var sinon = require('sinon');

require('chai').should();

function calculateHypotenuse(x, y, callback) {
  callback(null, Math.sqrt(x*x + y*x));
}

describe("When the user calculates the hypotenuse", function(){
  it("should execute the callback passed as argument", function() {
    var callback = sinon.spy();
    calculateHypotenuse(3, 3, callback);
    callback.called.should.be.true;
  });
});
```

我们再一次使用 Mocha 来运行脚本，并使用 Chai 的 `should` 接口来验证测试的结果，如下图所示：



之前的脚本中有一行需要重点关注：

```
var callback = sinon.spy();
```

此处，我们创建了一个 spy，并将它作为回调 callback 注入到函数中。由 Sinon.JS 创建的这个函数事实上并不是一个函数，而是一个携带了很多有趣信息的对象。Sinon.JS 利用 JavaScript 的动态特性实现了这一功能。你可以通过 `console.log()`将该对象输出到控制台，从而能观察到它的内部结构。

Sinon.JS 的另一个强大功能是 stub。stub 与 mock 非常相似（在 JavaScript 中具有相同的实际效果），我们可以通过它来伪造函数从而模拟出期望的返回值：

```
var sinon = require('sinon');
var expect = require('chai').expect;

function rollDice() {
  return -1 * Math.floor(Math.random() * 6) + 1;
}
describe("When rollDice gets called", function() {
  it("Math#random should be called with no arguments", function() {
    sinon.stub(Math, "random");
    rollDice();
    console.log(Math.random.calledWith());
  });
})
```

在这个例子里，我们为 `Math#random` 生成了 stub，起到的效果类似采用一个空函数

（它不会发起实际的调用）来对原函数进行覆盖，该空函数记录并统计了其自身的被调用情况。

对于前面的代码，有一个需要关注的地方：我们并没有恢复原来的 `random()` 方法，这样做相当危险。这会带来巨大的副作用，其他测试所看到的 `Math#random` 也都将是我们生成的 stub，而不是原来的函数，从而导致编写出根据无效信息生成的测试代码。

为了防止这种情况发生，我们需要利用 Mocha 的 `before()` 和 `after()` 方法：

```
var sinon = require('sinon');
var expect = require('chai').expect;

var sinon = require('sinon');
var expect = require('chai').expect;

function rollDice() {
  return -1 * Math.floor(Math.random() * 6) + 1;
}
describe("When rollDice gets called", function() {

  it("Math#random should be called with no arguments", function() {
    sinon.stub(Math, "random");
    rollDice();
    console.log(Math.random.calledWith());
  });
after(function(){
    Math.random.restore();
  });
});
```

如果你关注到了上述粗体部分的代码的话，就会发现 Sinon.JS 在 `it` 块中将 stub 的方法恢复成了原方法，这样一来，如果另一个 `describe` 块需要使用 `Math#random` 的话，它看到的将是原方法而不再是 stub。

方法 `before()` 和 `after()` 对于测试上下文的创建和销毁来说是很有帮助的。然而，你需要谨慎处理好它们执行的有效范围，不然可能会导致测试间的互相影响。

Mocha 拥有多种有关 before 和 after 的不同使用方式，如下所述。

- `before(callback)`：在当前范围之前执行（对于上述代码来说，会在 `describe` 块之前执行）。
- `after(callback)`：在当前范围之后执行（对于上述代码来说，会在 `describe` 块之后执行）。
- `beforeEach(callback)`：在当前范围内每个元素开始前执行（对于上述代码来说，会在每个 `it` 之前执行）。
- `afterEach(callback)`：在当前范围内每个元素结束后执行（对于上述代码来说，会在每个 `it` 之后执行）。

Sinon.JS 的另一个有趣的功能是对时间的控制。有些测试需要执行周期性的任务，或者是在某些事件发生之后的一定时间内做出响应。通过 Sinon.JS，我们可以将时间作为测试的参数之一来指定：

```
var sinon = require('sinon');
var expect = require('chai').expect

function areWeThereYet(callback) {

  setTimeout(function() {
    callback.apply(this);
  }, 10);

}

var clock;

before(function(){
  clock = sinon.useFakeTimers();
});

it("callback gets called after 10ms", function () {
  var callback = sinon.spy();
  var throttled = areWeThereYet(callback);
```

```
    areWeThereYet(callback);

    clock.tick(9);
    expect(callback.notCalled).to.be.true;

    clock.tick(1);
    expect(callback.notCalled).to.be.false;
  });

  after(function(){
    clock.restore();
  });
```

正如你所看到的，我们现在可以控制测试里的相关时间了。

测试真正的微服务

现在，是时候来测试一下真正的微服务了，从而让我们对完整的测试套件有一个全景的认识。

我们打算在微服务中使用 Express，该微服务可以过滤掉输入文本中被搜索引擎称为停用词的内容（stop words）：小于三个字符的词或被禁用的词。

来看一下代码：

```
var _ = require('lodash');
var express = require('express');

var bannedWords = ["kitten", "puppy", "parrot"];

function removeStopWords (text, callback) {
  var words = text.split(' ');
  var validWords = [];
  _(words).forEach(function(word, index) {
    var addWord = true;

    if (word.length < 3) {
      addWord = false;
    }
```

```
    if(addWord && bannedWords.indexOf(word) > -1) {
      addWord = false;
    }

    if (addWord) {
      validWords.push(word);
    }

    // 最后一轮迭代:
    if (index == (words.length - 1)) {
      callback(null, validWords.join(" "));
    }
  });
}
var app = express();

app.get('/filter', function(req, res) {
  removeStopWords(req.query.text, function(err, response){
    res.send(response);
  });
});

app.listen(3000, function() {
  console.log("app started in port 3000");
});
```

正如你所看到的，这个服务相当小，所以它是用以说明如何编写单元测试、集成测试和 E2E 测试的绝佳例子。在这个例子里，正如之前所提到的，E2E 测试和集成测试几乎是相同的，它们都通过 REST API 来测试服务，这与以端到端的视角来测试系统是等同的。

TDD——测试驱动开发

我们的服务已经完成开发并开始运行了。然而，当正想对它进行单元测试的时候，却发现了一些问题：

- 我们想要进行单元测试的函数对外部的主.js 文件来说是不可见的。
- 服务器端代码与功能性代码强烈耦合，缺乏良好的内聚。

TDD 便是此时解救这一困境的良药；我们应该常常问问自己："编写软件的时候我们会如何测试这个函数？"这并不意味着我们应该为了达到特定的测试目的去修改软件，但是一旦你在测试程序的某一部分时出现了问题，大多数时候都应该查看一下该程序的内聚性与耦合度，因为它们通常会反映出问题所在。让我们来看看下面的文件：

```
var _ = require('lodash');
var express = require('express');

module.exports = function(options) {
  bannedWords = [];
  if (typeof options !== 'undefined') {
    console.log(options);
    bannedWords = options.bannedWords || [];
  }

  return function bannedWords(text, callback) {
    var words = text.split(' ');
    var validWords = [];
    _(words).forEach(function(word, index) {
      var addWord = true;

      if (word.length < 3) {
        addWord = false;
      }

      if(addWord && bannedWords.indexOf(word) > -1) {
        addWord = false;
      }

      if (addWord) {
        validWords.push(word);
      }

      // 最后一轮迭代
      if (index == (words.length - 1)) {
        callback(null, validWords.join(" "));
```

```
        }
    });
  }
}
```

在我看来，该文件本身就是一个可高度重用且内聚性良好的模块：

- 我们可以在任意地方导入并使用它（甚至是在浏览器中）。
- 禁用词可在创建模块时进行注入（对于测试来说非常有用）。
- 与应用的代码没有掺和在一起。

以这种方式来组织代码的话，我们的应用代码将会是下面这个样子：

```
var _ = require('lodash');
var express = require('express');

var removeStopWords = require('./remove-stop-words')({bannedWords:
  ["kitten", "puppy", "parrot"]});

var app = express();

app.get('filter', function(req, res) {
  res.send(removeStopWords(req.query.text));
});

app.listen(3000, function() {
  console.log("app started in port 3000");
});
```

正如你所看到的，我们将业务单元（专注于业务逻辑的函数）与运维单元（服务器的装配逻辑）分离开了。

正如之前所提到的，我并不是一个热衷于代码未动、测试先行的人。但是我认为测试还是要伴随代码一起编写，同时也请记住之前提到的那些问题。

在一些公司中似乎有一些推手在推动 TDD 方法论的实施，但是这也可能会导致显著的效率低下问题，尤其是当一些业务尚不明朗的时候（通常项目有 90%的时间处于这种状态），我们会在开发的过程中面临大量的变化。

单元测试

现在，我们的代码已经逐步成型，是时候来对函数进行单元测试了。我们将使用 Mocha 和 Chai 来完成如下的任务：

```
var removeStopWords = require('./remove-stop-words')({bannedWords:
  ["kitten", "parrot"]});

var chai = require('chai');
var assert = chai.assert;
chai.should();
var expect = chai.expect;

describe('When executing "removeStopWords"', function() {

  it('should remove words with less than 3 chars of length',
    function() {
    removeStopWords('my small list of words', function(err,
      response) {
      expect(response).to.equal("small list words");
    });
  });

  it('should remove extra white spaces', function() {
    removeStopWords('my small      list of words', function(err,
      response) {
      expect(response).to.equal("small list words");
    });
  });

  it('should remove banned words', function() {
    removeStopWords('My kitten is sleeping', function(err,
      response) {
      expect(response).to.equal("sleeping");
    });
  });

  it('should not fail with null as input', function() {
```

```
    removeStopWords(null, function(err, response) {
      expect(response).to.equal("small list words");
    });
  });

  it('should fail if the input is not a string', function() {
    try {
      removeStopWords(5, function(err, response) {});
      assert.fail();
    }
    catch(err) {
    }
  });
});
```

正如你所看到的，我们已经覆盖了几乎每一个测试案例以及应用中的各个分支，但是测试的代码覆盖率到底是多少呢？

到目前为止，我们只是提到了代码覆盖率，但是却没有真正评测过。我们将通过一个叫作 Istanbul 的工具来评测一下测试覆盖率：

```
npm install -g istanbul
```

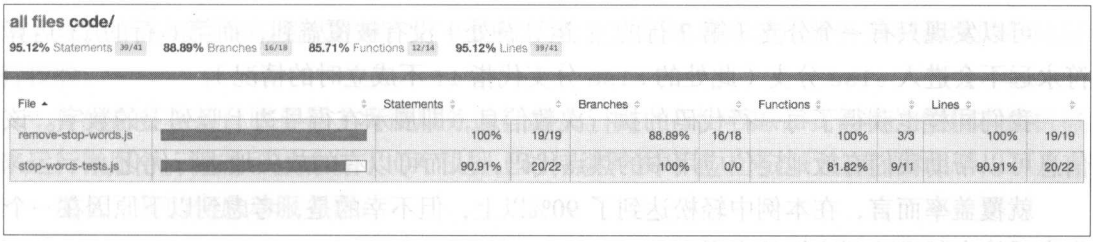
通过上述命令我们完成了 Istanbul 的安装，现在可以通过运行如下命令来生成覆盖率报告了：

```
istanbul cover _mocha my-tests.js
```

该命令将会生成一份类似如下图所示的输出报告：

```
=============================== Coverage summary ===============================
Statements   : 95.12% ( 39/41 )
Branches     : 88.89% ( 16/18 )
Functions    : 85.71% ( 12/14 )
Lines        : 95.12% ( 39/41 )
================================================================================
```

同时还会生成一份 HTML 格式的覆盖率报告，报告中指出了哪些行、函数、分支及语句没有被覆盖到，如下屏幕截图所示：

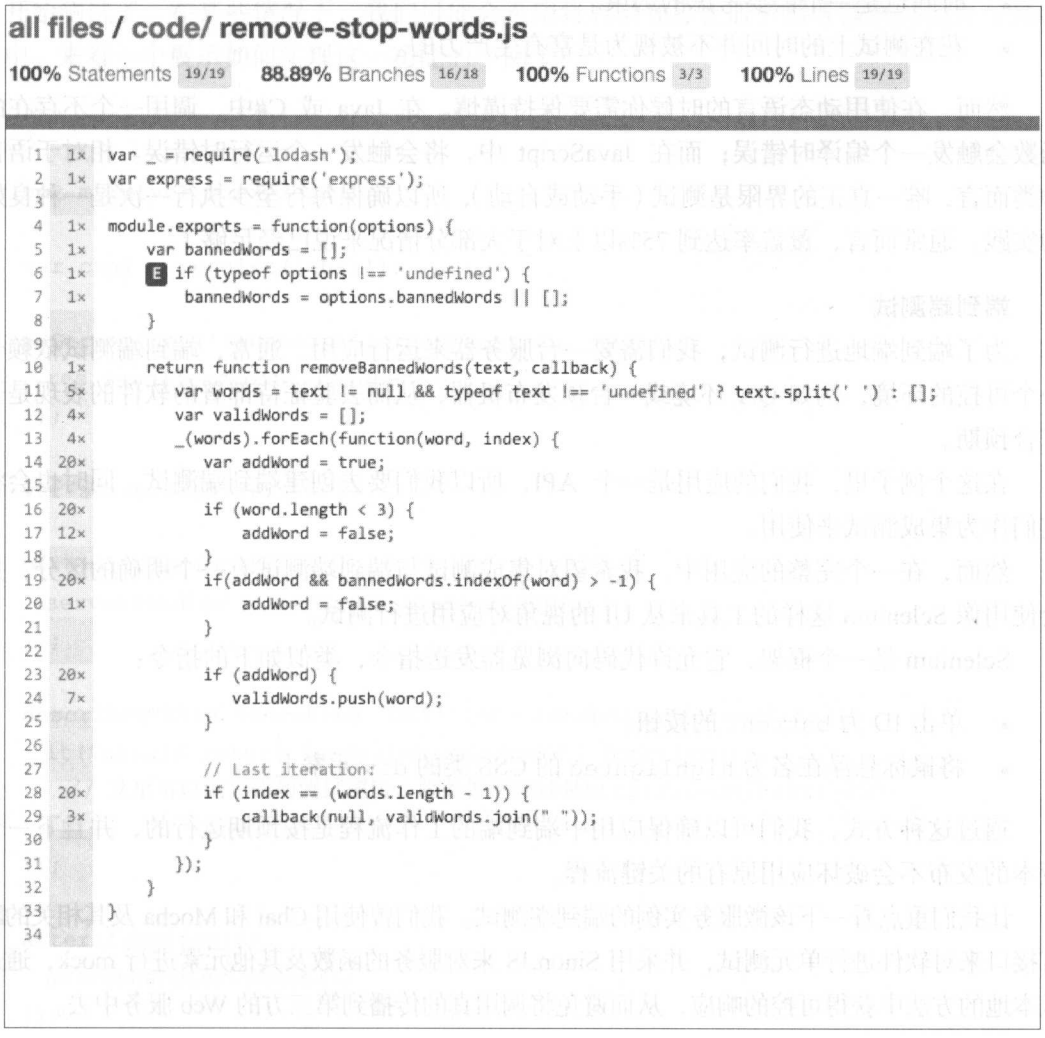


all files code/

95.12% Statements 39/41 88.89% Branches 16/18 85.71% Functions 12/14 95.12% Lines 39/41

File	Statements		Branches		Functions		Lines	
remove-stop-words.js	100%	19/19	88.89%	16/18	100%	3/3	100%	19/19
stop-words-tests.js	90.91%	20/22	100%	0/0	81.82%	9/11	90.91%	20/22

正如你所看到的，一切非常顺利，我们的代码（而不是测试）确确实实被覆盖到了。尤其是当我们进一步查看代码文件的具体报告时，如下图所示：



all files / code/ remove-stop-words.js

100% Statements 19/19 88.89% Branches 16/18 100% Functions 3/3 100% Lines 19/19

```
1  1×  var _ = require('lodash');
2  1×  var express = require('express');
3
4  1×  module.exports = function(options) {
5  1×      var bannedWords = [];
6  1×      E if (typeof options !== 'undefined') {
7  1×          bannedWords = options.bannedWords || [];
8          }
9
10 1×      return function removeBannedWords(text, callback) {
11 5×          var words = text != null && typeof text !== 'undefined' ? text.split(' ') : [];
12 4×          var validWords = [];
13 4×          _(words).forEach(function(word, index) {
14 20×             var addWord = true;
15
16 20×             if (word.length < 3) {
17 12×                 addWord = false;
18             }
19 20×             if(addWord && bannedWords.indexOf(word) > -1) {
20 1×                  addWord = false;
21             }
22
23 20×             if (addWord) {
24 7×                 validWords.push(word);
25             }
26
27             // Last iteration:
28 20×             if (index == (words.length - 1)) {
29 3×                 callback(null, validWords.join(" "));
30             }
31         });
32     }
33 }
34
```

可以发现只有一个分支（第 7 行的 || 运算符处）没有被覆盖到，而第 6 行的 `if` 运算符永远不会进入 `else` 分支（此处的 `else` 分支代指 `if` 不成立时的情况）。

我们同样也获得了每一行代码的执行次数信息，即展示在行号边上竖列上的数字。该信息可以帮助我们有效地定位应用中的热点代码，从而可以有的放矢地进行优化。

就覆盖率而言，在本例中轻松达到了 90%以上，但不幸的是，考虑到以下原因在一个生产系统中很难达到这样的水平：

- 代码更加复杂。
- 时间也是一个需要考虑的约束。
- 花在测试上的时间并不被视为是富有生产力的。

然而，在使用动态语言的时候你需要保持谨慎。在 Java 或 C#中，调用一个不存在的函数会触发一个编译时错误；而在 JavaScript 中，将会触发一个运行时错误。相对于语言种类而言，唯一真正的界限是测试（手动或自动），所以确保每行至少执行一次是一种良好的实践。通常而言，覆盖率达到 75%以上对于大部分情况来说已经足够了。

端到端测试

为了端到端地进行测试，我们需要一台服务器来运行应用。通常，端到端测试依赖于一个可控的环境，例如 QA 环境或一台预发布机器，从而去验证待部署的软件的表现是否符合预期。

在这个例子里，我们的应用是一个 API，所以我们要去创建端到端测试，同时也会将它们作为集成测试来使用。

然而，在一个完整的应用中，我希望对集成测试与端到端测试有一个明确的区分，并会使用像 Selenium 这样的工具来从 UI 的视角对应用进行测试。

Selenium 是一个框架，它允许代码向浏览器发送指令，类似如下的指令：

- 单击 ID 为 `button1` 的按钮。
- 将鼠标悬浮在名为 `highlighted` 的 CSS 类的 `div` 元素上。

通过这种方式，我们可以确保应用中端到端的工作流程是按预期运行的，并且下一个版本的发布不会破坏应用原有的关键流程。

让我们重点看一下该微服务实例的端到端测试。我们曾使用 Chai 和 Mocha 及其相关的断言接口来对软件进行单元测试，并采用 Sinon.JS 来对服务的函数及其他元素进行 mock，通过从本地的方法中获得可控的响应，从而避免将调用真的传播到第三方的 Web 服务中去。

现在，在端到端的测试计划中，我们确实希望发起请求来调用服务并对获取的响应进行校验。

首先需要在某台服务器上将微服务运行起来。出于方便考虑，我们将先使用本地机器来运行微服务，但是可以在位于 QA 机器上的持续开发环境中来执行这些测试。

那么，我们先启动服务器：

```
node stop-words.js
```

出于方便考虑，我手动调用了 stop-words.js 脚本。一旦服务器运行起来，我们便可以开始测试了。在某些情况下，我们可能会希望将启动和停止服务器等流程都包含在测试之中。来看一个展示如何实现这一功能的小例子：

```
var express = require('express');

var myServer = express();

var chai = require('chai');

myServer.get('/endpoint', function(req, res){
  res.send('endpoint reached');
});

var serverHandler;

before(function(){
  serverHandler = myServer.listen(3000);
});

describe("When executing 'GET' into /endpoint", function(){
  it("should return 'endpoint reached'", function(){
    // 这里可以编写你的测试逻辑，服务器的地址是http://localhost:3000
  });
});

after(function(){
  serverHandler.close();
});
```

正如你所看到的，Express 提供了一个能以编程方式创建服务器的句柄，从而可以结合对 before() 和 after() 函数的利用来满足这一需求。

在该例子中，我们假定服务器已经开始运行了。为了发起请求，我们需要使用一个名为 request 的库来向服务器发起调用。

安装的方式跟之前一样，我们采用 npm install request 命令来安装该库。一旦安装完毕，便可以好好体验一下这个让人惊艳的库了：

```
var chai = require('chai');
var chaiHttp = require('chai-http');
var expect = chai.expect;
chai.use(chaiHttp);

describe("when we issue a 'GET' to /filter with text='aaaa bbbb
  cccc'", function(){
  it("should return HTTP 200", function(done) {
    chai.request('http://localhost:3000')
      .get('/filter')
      .query({text: 'aa bb ccccc'}).end(function(req, res){
        expect(res.status).to.equal(200);
        done();
      });
  });
});

describe("when we issue a 'GET' to /filter with text='aa bb
  ccccc'", function(){
  it("should return 'ccccc'", function(done) {
    chai.request('http://localhost:3000')
      .get('/filter')
      .query({text: 'aa bb ccccc'}).end(function(req, res){
        expect(res.text).to.equal('ccccc');
        done();
      });
  });
});

describe("when we issue a 'GET' to /filter with text='aa bb cc'",
  function(){
```

```
  it("should return ''", function(done) {
    chai.request('http://localhost:3000')
      .get('/filter')
      .query({text: 'aa bb cc'}).end(function(req, res){
        expect(res.text).to.equal('');
        done();
      });
  });
});
```

通过这些简单的测试，我们已经解决了服务器的测试问题，从而确保应用中每一个部分都会被执行到。

这里有一个部分比较特殊，之前我们从未使用过：

```
  it("should return 'ccccc'", function(done) {
    chai.request('http://localhost:3000')
      .get('/filter')
      .query({text: 'aa bb ccccc'}).end(function(req, res){
        expect(res.text).to.equal('ccccc');
        done();
      });
  });
```

如果关注一下粗体部分的代码，你会发现有一个名为 done 的回调函数。该回调函数拥有一个使命：在它被调用前阻止测试结束，从而让 HTTP 请求有时间得到执行从而返回合适的值。请记住，Node.js 是异步的，它没有让线程阻塞直到某个操作完成的功能。

除此之外，我们还使用了一个由 chai-http 引入的 DSL 来构建 get 请求。

通过该语言可以构建一个由一系列组合构成的请求，如下例所示：

```
chai.request('http://mydomain.com')
  .post('/myform')
  .field('_method', 'put')
  .field('username', 'dgonzalez')
  .field('password', '123456').end(...)
```

在该请求中，我们提交了一个形似登录框的表单，所以在 end() 函数中，可以对服务器返回的内容进行断言测试。

通过 chai-http，可以拼接出任意数量的组合来测试 API。

人工测试——迫不得已却又不可或缺

不管你在自动化测试上花费了多少精力，通常一定数量的人工测试还是必不可少的。

有时，为了在开发 API 的时候观察从客户端发往服务端的消息，需要进行人工测试。而在其他一些时候，我们想要服务端能命中并处理我们预加工过的请求，促使软件按期望的方式执行，那么人工测试也能满足需求。

对于第一种情况，我们将利用 Node.js 及其动态特性来构建一个代理，该代理可以嗅探到所有的请求并将它们以日志的方式记录到终端中，以便于我们调试请求的处理情况，这一技术可用于观察两个微服务之间的通信，从而在无须中断处理流程的情况下观察到请求的处理进展。

对于第二种情况，我们将使用一款叫作 Postman 的软件来以一种可控的方式向服务器发起请求。

构建代理来调试微服务

我第一次接触 Node.js 是因为恰好遇上这样一个问题：两台服务器互相向对方发送消息，产生了异常的行为却没有找到任何明显的原因。

这是一个有着很多可选解决方案且又非常常见的问题（大体上跟中间人代理差不多），但是我们将展示的是 Node.js 有多么强大：

```
var http = require('http');
var httpProxy = require('http-proxy');
var proxy = httpProxy.createProxyServer({});

http.createServer(function(req, res) {
  console.log(req.rawHeaders);
  proxy.web(req, res, { target: 'http://localhost:3000' });
}).listen(4000);
```

如果你还记得前面小节的内容，那么就会知道 stop-words.js 程序是运行在 3000 端口的。而我们目前的代码所做的就是使用 http-proxy 来创建一个代理，并将发往 4000 端口的请求的头部记录到控制台上，然后打通隧道来将原请求路由到 3000 端口。

如果我们已采用 npm install 命令完成了所有依赖的安装，那么在项目根目录中运行程序时将会看到该代理是如何有效地将请求记录下来，并将其路由到目标主机的：

```
curl http://localhost:4000/filter?text=aaa
```

该命令将会产生如下输出：

```
➜  code  node proxy.js
[ 'Host',
  'localhost:4000',
  'User-Agent',
  'curl/7.43.0',
  'Accept',
  '*/*' ]
```

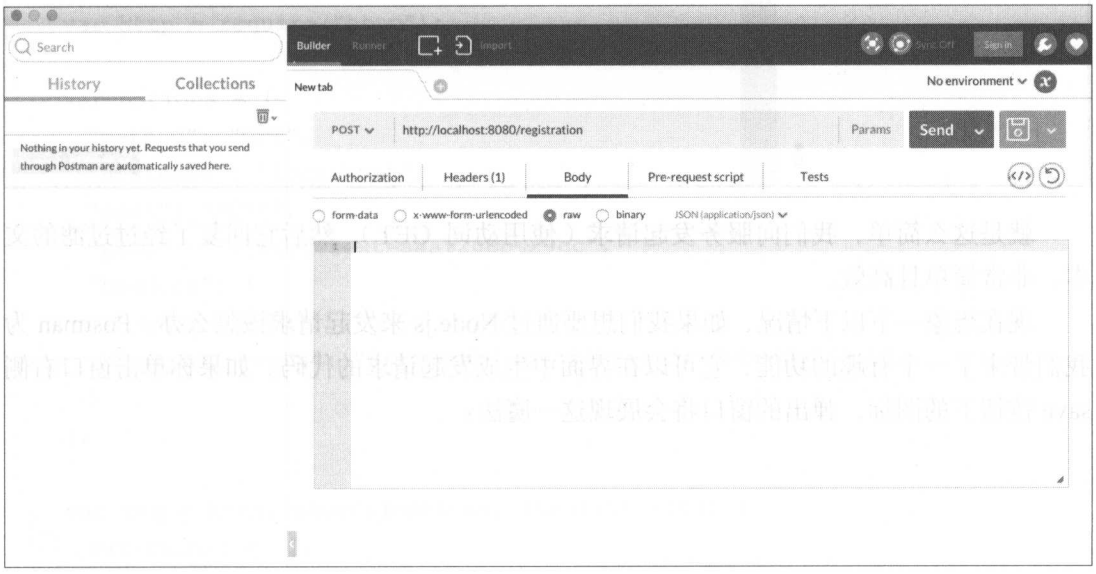
这个例子是如此简单，该小型代理几乎可以部署在我们的微服务之间的任何地方，同时也让我们能观察到有关网络处理的非常有价值的信息。

Postman

在所有能在网络上找到的用于测试 API 的软件中，Postman 才是我的最爱。它起初是作为 Google Chrome 浏览器的一个扩展开发的，而现如今，它已经是一款构建在 Chrome 运行时上的独立应用了。

你可以在 Chrome 的 Web 商店中找到它，它拥有免费版（所以你无须为此支付任何费用），但同时也拥有一个供团队使用的版本，该版本则具有更多的高级功能且需要付费。

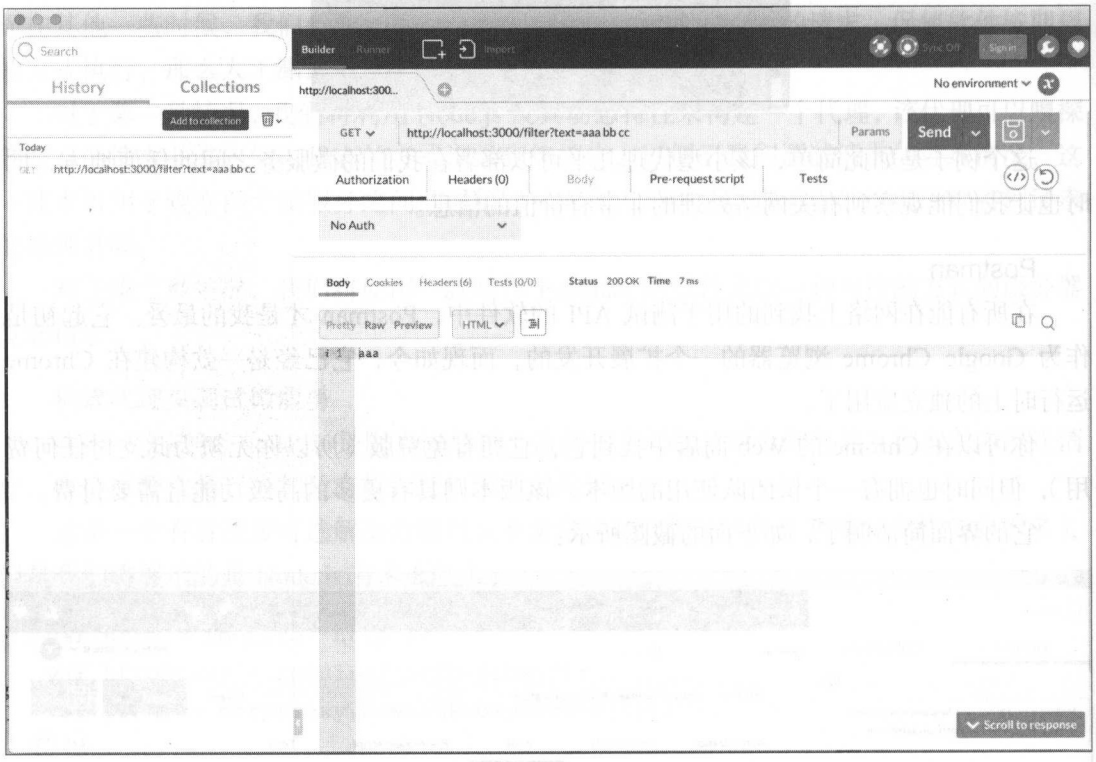
它的界面简洁明了，如下面的截图所示：



在页面左侧，可以看到请求的历史记录以及请求的集合，这对参与长期项目的开发者来说非常便利，我们可以用它来构建一些复杂的请求。

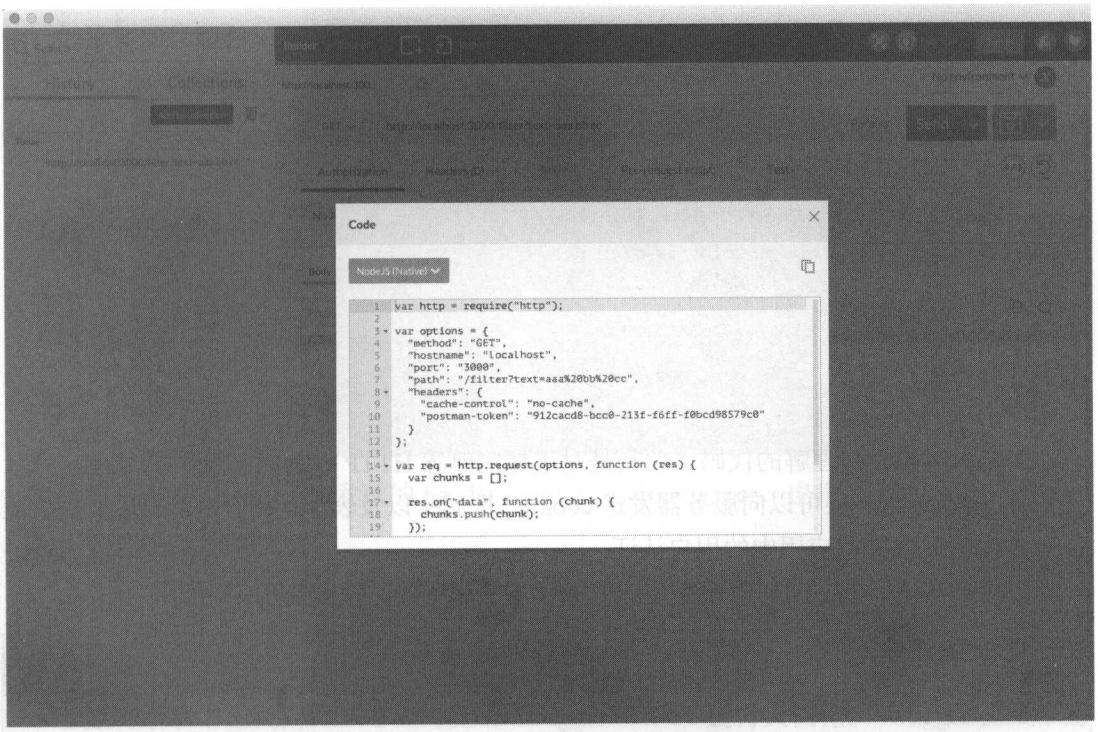
接着我们将再次使用 `stop-words.js` 微服务来展示 Postman 的强大威力。

因此，第一件事就是确保微服务已经运行起来，一旦服务启动，我们便可以通过 Postman 来向它发起请求，如下面的截图所示：



就是这么简单，我们向服务发起请求（使用动词 GET），然后它回复了经过过滤的文本，非常简单且高效。

现在想象一下以下情况，如果我们想要通过 Node.js 来发起请求该怎么办。Postman 为我们带来了一个有趣的功能，它可以在界面中生成发起请求的代码。如果你单击窗口右侧 save 按钮下的图标，弹出的窗口将会展现这一魔法：



让我们来看看生成的代码：

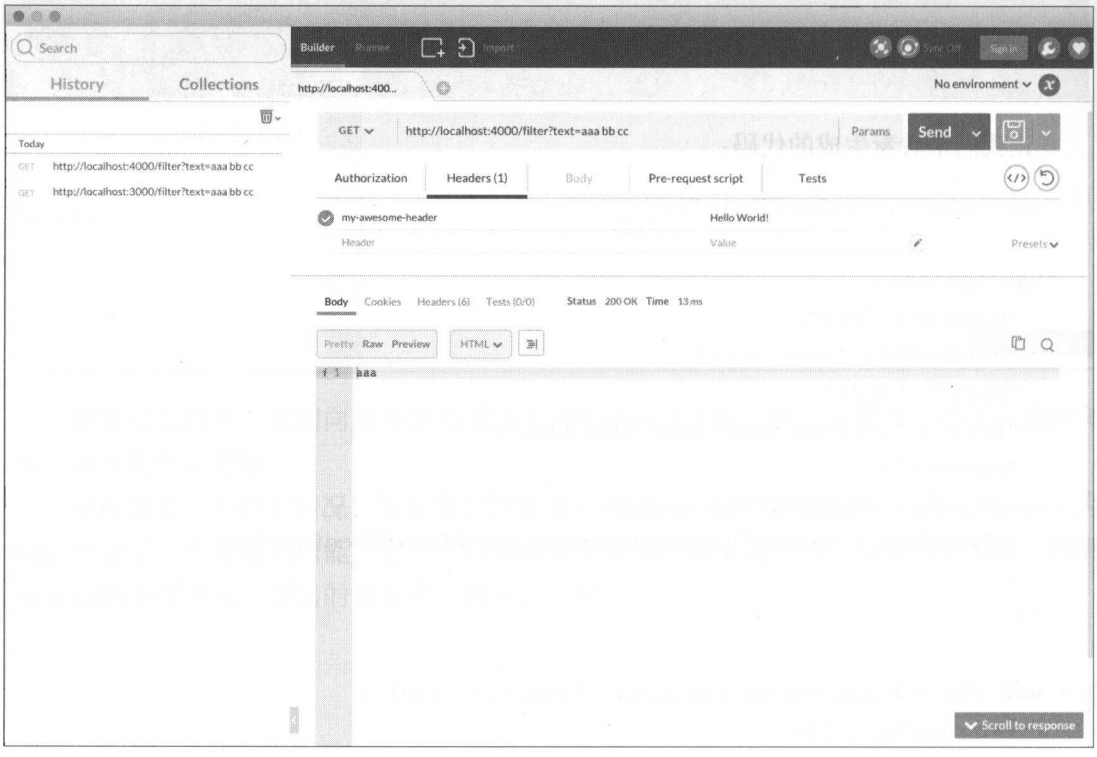
```
var http = require("http");

var options = {
  "method": "GET",
  "hostname": "localhost",
  "port": "3000",
  "path": "/filter?text=aaa%20bb%20cc",
  "headers": {
    "cache-control": "no-cache",
    "postman-token": "912cacd8-bcc0-213f-f6ff-f0bcd98579c0"
  }
};

var req = http.request(options, function (res) {
  var chunks = [];
```

```
    res.on("data", function (chunk) {
      chunks.push(chunk);
    });

    res.on("end", function () {
      var body = Buffer.concat(chunks);
      console.log(body.toString());
    });
  });

  req.end();
```

这是一段很容易理解的代码，尤其是对于那些熟悉 HTTP 库的人来说。

通过 Postman，还可以向服务器发送 cookie、header 以及表单，从而可以通过发送认证令牌和 cookie 来模拟应用中的用户认证。

让我们将请求重定向到前面小节创建的代理，如下面的截图所示：



如果你的代理和 stop-words.js 微服务都正常运行着，你将在代理中看到如下输出：

```
→ code  node proxy.js
[ 'Host',
  'localhost:4000',
  'User-Agent',
  'curl/7.43.0',
  'Accept',
  '*/*' ]
[ 'Host',
  'localhost:4000',
  'Connection',
  'keep-alive',
  'Cache-Control',
  'no-cache',
  'my-awesome-header',
  'Hello World!',
  'User-Agent',
  'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.116 Safari/537.36',
  'Postman-Token',
  '9381e356-2d24-8ffe-82c6-6b16c837af18',
  'Accept',
  '*/*',
  'Accept-Encoding',
  'gzip, deflate, sdch',
  'Accept-Language',
  'es-ES,es;q=0.8,en;q=0.6' ]
```

我们通过 Postman 一起发送的 header：my-awesome-header，也将展示在原始的 header 列表中。

对微服务进行文档化

在这一小节中，我们将学习如何使用 Swagger 对 API 进行文档化。Swagger 是一个遵守开放 API 标准的 API 管理工具，所以它也可以算是 API 创建者们共通的语言。我们将会讨论如何编写定义以及讲述统一资源描述方式的重要性。

采用 Swagger 对 API 进行文档化

文档化常常是一个烦人的问题。无论你多么努力地尝试，文档通常最终都会过时。幸运的是，在过去的几年里，有一股潮流推动着 REST API 的高质量文档化。

API 管理工具在其中扮演了重要的角色，而 Swagger 正是一个在这方面非常有意思的平台。Swagger 不仅是一个文档化的模块，其管理 API 的方式也独具特色，可以让你对自己的工作拥有一个整体的视图。

让我们先安装上 Swagger：

```
npm install -g swagger
```

上述命令已经在系统级别安装了 Swagger，所以 swagger 已经成为系统的一个新命令。现在，我们来使用它创建一个项目：

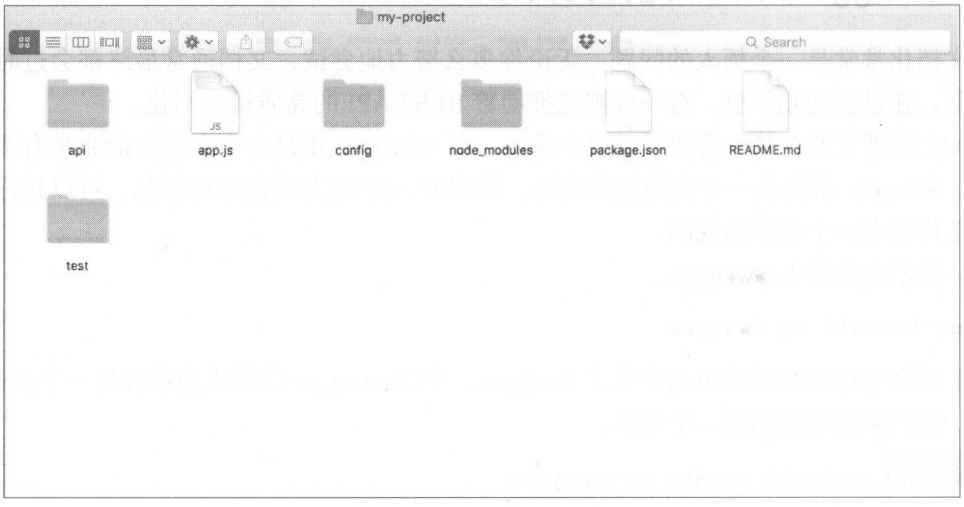
```
swagger project create my-project
```

该命令将会提示你选择不同的 Web 框架。我们将会选择 Express，这是一个我们已经使用过的框架。上述命令的输出如下面的截图所示：

```
➜ Documents  swagger project create my-project
? Framework? express
Project my-project created in /Users/dgonzalez/Documents/my-project
Running "npm install"...
npm
 WARN
deprecated lodash@2.4.2: lodash@<3.0.0 is no longer maintained. Upgrade to lodash@^4.0.0.

should@7.1.1 node_modules/should
├── should-type@0.2.0
├── should-equal@0.5.0
└── should-format@0.3.1

express@4.13.4 node_modules/express
├── escape-html@1.0.3
├── array-flatten@1.1.1
├── utils-merge@1.0.0
├── cookie-signature@1.0.6
├── merge-descriptors@1.0.1
├── methods@1.1.2
├── fresh@0.3.0
├── range-parser@1.0.3
├── vary@1.0.1
├── path-to-regexp@0.1.7
├── cookie@0.1.5
├── parseurl@1.3.1
├── content-type@1.0.1
├── etag@1.7.0
├── content-disposition@0.5.1
├── serve-static@1.10.2
├── depd@1.1.0
├── qs@4.0.0
├── on-finished@2.3.0 (ee-first@1.1.1)
├── debug@2.2.0 (ms@0.7.1)
├── proxy-addr@1.0.10 (forwarded@0.1.0, ipaddr.js@1.0.5)
├── send@0.13.1 (statuses@1.2.1, ms@0.7.1, destroy@1.0.4, mime@1.3.4, http-errors@1.3.1)
├── finalhandler@0.4.1 (unpipe@1.0.0)
├── type-is@1.6.12 (media-typer@0.3.0, mime-types@2.1.10)
└── accepts@1.2.13 (negotiator@0.5.3, mime-types@2.1.10)

supertest@1.2.0 node_modules/supertest
├── methods@1.1.2
└── superagent@1.7.2 (component-emitter@1.2.0, cookiejar@2.0.6, reduce-component@1.0.1, extend@3.0.0, mime@1.3.4, formidable@1.0.17, qs@2.3.3, debug@2.2.0, readable-stream@1.0.27-1, form-data@0.2.0)

swagger-express-mw@0.1.0 node_modules/swagger-express-mw
└── swagger-node-runner@0.5.13 (debug@2.2.0, cors@2.7.1, config@1.19.0, lodash@3.10.1, js-yaml@3.5.3, bagpipes@0.0.6, swagger-tools@0.9.16)

Success! You may start your new app by running: "swagger project start my-project"
```

该截图展示了如何采用Swagger来启动项目

现在我们可以看到生成了一个新的叫作 `my-project` 的文件夹，如下图所示：



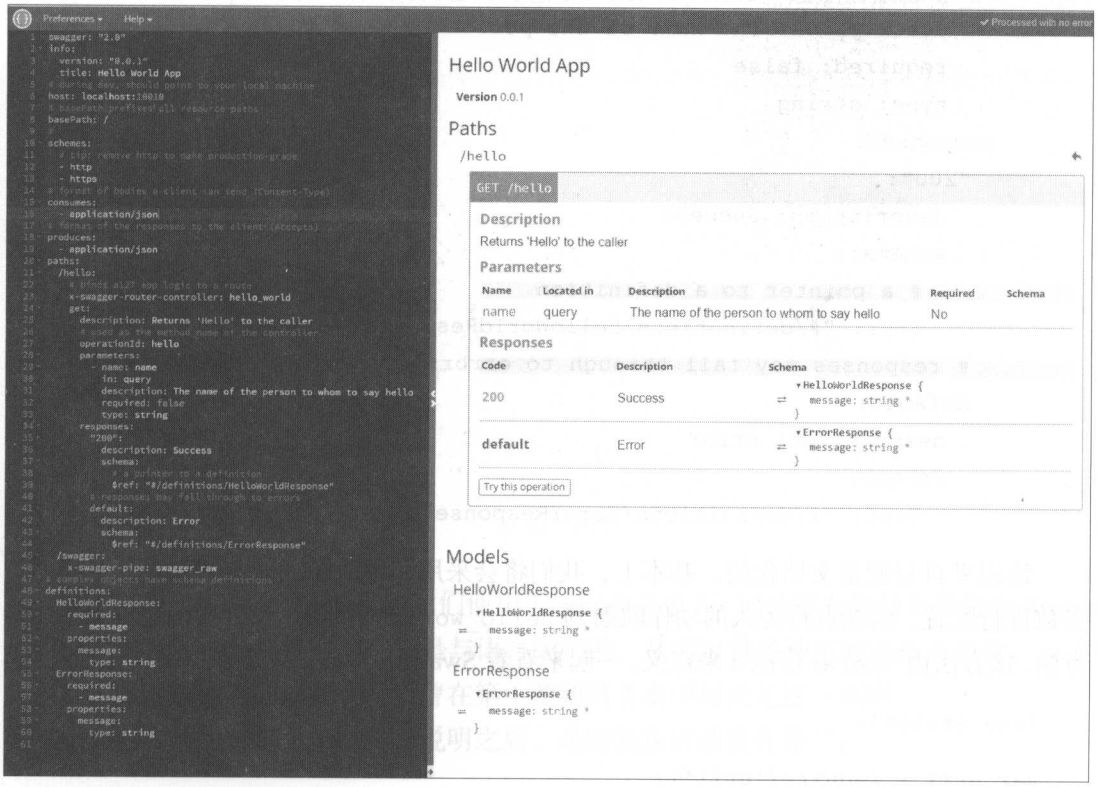
其结构不言自明，同时也符合一个 Node.js 应用的常见布局。

- `api`：此处存放着我们的 API 代码。
- `config`：所有的配置文件都存放在这里。
- `node_modules`：该文件夹中存放着应用所需的所有依赖。
- `test`：Swagger 将生成的测试代码放于此处，我们也可以在此添加自己的测试。

Swagger 拥有一项让人印象深刻的功能：一个内嵌的编辑器，让你可以对 API 的端点进行建模。为了运行该编辑器，我们要在生成的文件夹中运行下面的命令：

```
Swagger project edit
```

该命令将会在默认的浏览器中打开 Swagger 编辑器，该窗口大致如下所示：



Swagger 使用了“另一种标记语言（YAML）”。该语言与 JSON 非常相似，但是语法完全不同。

在下面的文档中，我们可以自定义很多内容，比如路径（在应用中起到路由的作用）。让我们来看看由 Swagger 生成的路径：

```
/hello:
  # binds a127 app logic to a route
  x-swagger-router-controller: hello_world
  get:
    description: Returns 'Hello' to the caller
    # used as the method name of the controller
    operationId: hello
    parameters:
      - name: name
        in: query
        description: The name of the person to whom to say hello
        required: false
        type: string
    responses:
      "200":
        description: Success
        schema:
          # a pointer to a definition
          $ref: "#/definitions/HelloWorldResponse"
      # responses may fall through to errors
      default:
        description: Error
        schema:
          $ref: "#/definitions/ErrorResponse"
```

该定义自身就是文档化的。基本上，我们将会采用一种声明式的方式来对端点使用的参数进行配置。该端点将传入的动作映射到 `hello_world` 控制器，并会明确定位到 `hello` 方法，该方法由 `operationId` 来定义。一起来看看 Swagger 将会在这个控制器中生成什么：

```
'use strict';

var util = require('util');

module.exports = {
  hello: hello
};
```

```
function hello(req, res) {
  var name = req.swagger.params.name.value || 'stranger';
  var hello = util.format('Hello, %s!', name);
  res.json(hello);
}
```

这段代码可以在项目的 api/controllers 文件夹中找到。正如你所看到的，这是一个非常标准的 Express 控制器，且已被打包成一个高内聚的模块。唯一让我们感到陌生的是 hello 函数的第一行，通过这行可以从 Swagger 中摘取出参数。我们将在稍后运行项目时再来回顾这些。

端点的第二部分是响应。我们可以看到，其中有两处定义：http 的状态码 200 对应 HelloWorldResponse，而其他状态码对应 ErrorResponse。这些对象都定义在下面的代码中：

```
definitions:
  HelloWorldResponse:
    required:
      - message
    properties:
      message:
        type: string
  ErrorResponse:
    required:
      - message
    properties:
      message:
        type: string
```

这一点确实很有意思，虽然我们使用了一门动态语言，但是所有的约定都是由 Swagger 定义的，以至于我们拥有的定义是与语言无关的，从而可供各种不同的技术消费。这便是对技术多样性原则的体现，我们曾在第 1 章和第 2 章中讨论过这一原则。

在对如何完成定义工作进行说明之后，是时候来启动服务器了：

```
swagger project start
```

该命令的输出与下面的代码非常相似：

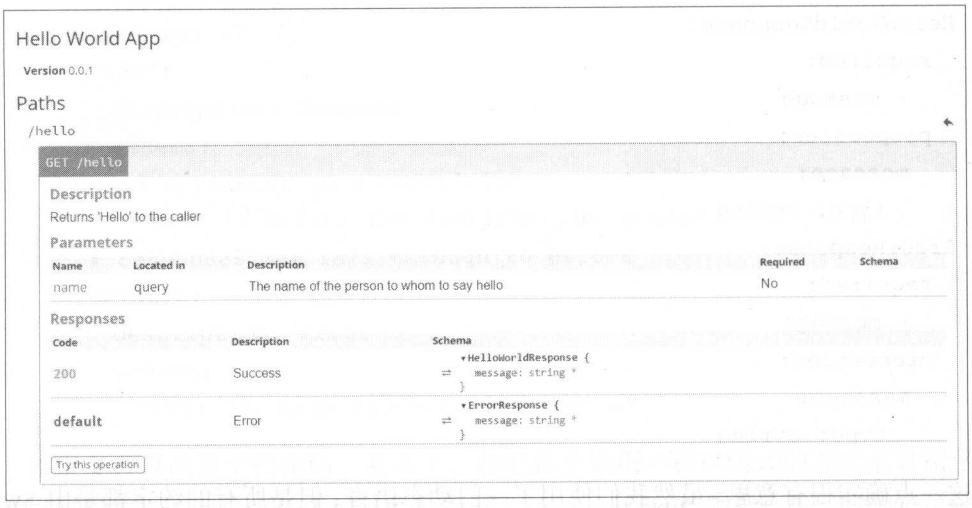
```
Starting: C:\my-project\app.js...
  project started here: http://localhost:10010/
  project will restart on changes.
  to restart at any time, enter `rs`
try this:
curl http://127.0.0.1:10010/hello?name=Scott
```

现在，如果遵照上述输出中提到的指令并执行 curl 命令，我们将会得到如下输出：

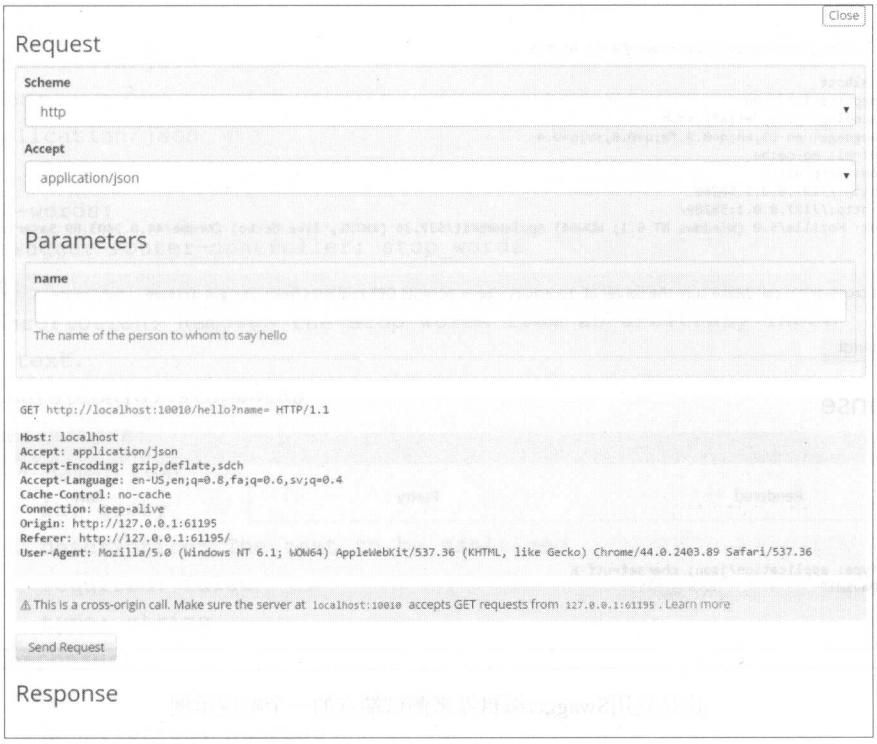
```
curl http://127.0.0.1:10010/hello?name=David
"Hello David!"
```

Swagger 将名为 name 的查询参数与 YAML 定义中指定的 Swagger 参数进行了绑定。这听上去可能很糟，因为我们将软件与 Swagger 耦合到了一起，但是这样做却给你带来了很大的好处：Swagger 允许你通过编辑器来测试端点，来看看这一点是怎么做到的。

在编辑器的左侧，可以看到一个标有“Try this operation”字样的按钮，如下图所示：



一旦你单击它，它将会向你呈现出一个表单，可以通过该表单来测试端点，如下图所示：

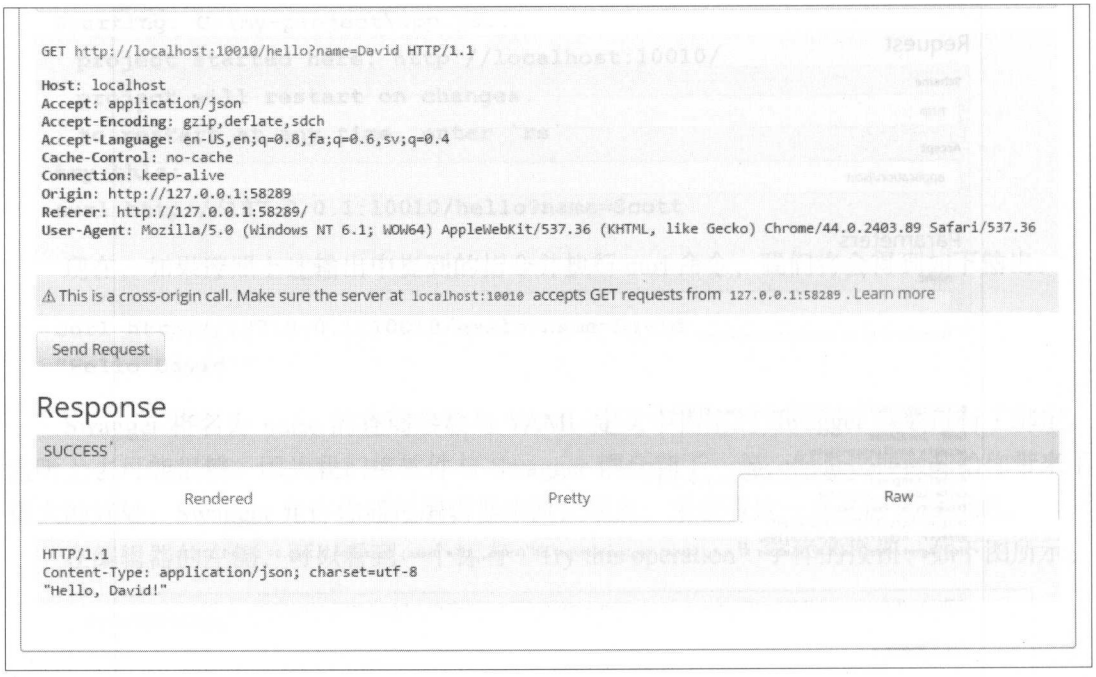


在这个表单底部，有一行关于跨源请求的提示信息。当我们在本机进行开发时无须担心该问题，然而，当使用 Swagger 编辑器来测试其他主机时就会遇到问题。

想了解更多关于这方面的信息可以访问以下链接：

`https://en.wikipedia.org/wiki/Cross-origin_resource_sharing`

为 `name` 参数填入一个值，然后单击“Send Request”按钮发送请求，如下图所示：



这是使用Swagger编辑器来测试端点的一个响应示例

请注意，为了让测试顺利工作，我们的应用服务器必须保持运行。

根据 Swagger 定义来生成项目

到现在为止，我们已经把玩过 Swagger 且用它生成了项目，接下来我们将根据 `swagger.yaml` 来生成项目。你将使用现已生成的这个项目作为起点，但是我们会向其添加一个新的端点：

```
swagger: "2.0"
info:
  version: "0.0.1"
  title: Stop Words Filtering App
host: localhost:8000
basePath: /
schemes:
  - http
  - https
```

```
consumes:
  - application/json
produces:
  - application/json
paths:
  /stop-words:
    x-swagger-router-controller: stop_words
    get:
      description: Removes the stop words from an arbitrary input
        text.
      operationId: stopwords
      parameters:
        - name: text
          in: query
          description: The text to be sanitized
          required: false
          type: string
      responses:
        "200":
          description: Success
          schema:
            $ref: "#/definitions/StopWordsResponse"
  /swagger:
    x-swagger-pipe: swagger_raw
definitions:
  StopWordsResponse:
    required:
      - message
    properties:
      message:
        type: string
```

该端点看上去应该并不陌生，因为你已经在本章之前的单元测试部分对它进行过测试。现在你应该已经知道 Swagger 编辑器有多酷了：它会根据你的输入提供实时反馈，同时会将你在 YAML 文件中的改动保存下来。

下一步就要从 `https://github.com/swagger-api/swagger-codegen` 下载代码生成器了。这是一个 Java 项目，所以我们需要 Java SDK 和 Maven 来构建它，如下所示：

```
mvn package
```

代码生成器是一个可以从 Swagger YAML 读取 API 定义，并构建出与我们所选语言对应的项目结构的工具，在本例中生成的便是一个 Node.js 项目。

在项目的根目录中执行前面的命令会对所有的子模块进行构建。现在，可以非常简单地在 `swagger-codegen` 文件夹的根目录中执行下面的命令：

```
java -jar modules/swagger-codegen-cli/target/swagger-codegen-cli.jar
generate -i my-project.yaml -l nodejs -o my-project
```

Swagger 代码生成器支持多种语言。这里的精妙之处在于，当将它用于微服务时，可用于定义接口进而使用最合适的技术来构建服务。

当你进入 `my-project` 文件夹时，会发现完整的项目结构已经置备妥当，就等你开始编写代码了。

小结

在本章中，我们学习了如何对微服务进行测试及文档化。而在软件开发中，这些活动通常会因为新功能的交付压力而被遗忘。但在我看来，不进行测试是一个充满风险的决策。我们必须在测试过量与不足之间找到平衡。通常，我们将会尝试找出单元测试、集成测试及端到端测试各自合适的占比。

我们同时还了解了人工测试以及用于对软件进行有效人工测试的工具（通常会有人工测试组件）。

另一个有趣的讨论点是关于文档化及 API 管理的。在本例中，我们了解了 Swagger，这基本上可以算得上是最流行的 API 管理工具了，它引领了开放 API 标准的创建。

如果你想在 API 的世界里走得更远（如果要构建实用且有效的 API 将会有很多内容需要学习），那或许应该去浏览一下 `http://apigee.com`。Apigee 是一家专业构建 API 的公司，同时它也为开发者和企业构建出更好的 API 提供了大量的工具。

7 微服务的监控

对服务器的监控向来是一个有争议的话题。我们通常将它划归到系统管理的范畴，软件工程师很少接触到它，但这样一来也让我们错失了由监控带来的巨大好处：快速响应失败的能力。通过对系统进行紧密监控，几乎可以在第一时间发现问题，并尽快解决问题，从而避免影响到客户对系统的使用。除了监控之外，另一个概念便是性能。通过了解系统在负载期内的表现，我们可以尽早做好容量规划。在本章中，我们将会讨论如何监控服务器，尤其是对微服务的监控，从而能更好地维护系统的稳定性。

在本章中，我们将会讨论以下话题：

- 服务监控
- 采用 PM2 和 Keymetrics 进行监控
- 监控指标
- 类人猿大军——来自 Netflix 的主动监控
- 吞吐量与性能的降级

服务监控

在监控微服务的时候，我们通常会关注几种不同类型的指标。第一组指标是有关硬件资源的，描述如下。

- **内存指标**：该指标表明系统中剩余多少内存，或者我们的应用消耗了多少内存。
- **CPU 使用率**：正如其名，该指标表明在某一时间点我们使用了多少 CPU。
- **磁盘使用率**：该指标表明了物理磁盘的 I/O 压力。

第二组是应用指标，如下所示：

- 单位时间内的错误数
- 单位时间内的调用数
- 响应时间

两组指标是互相关联的，硬件的问题会影响应用性能（反过来也一样），所以必须了解这两组指标。

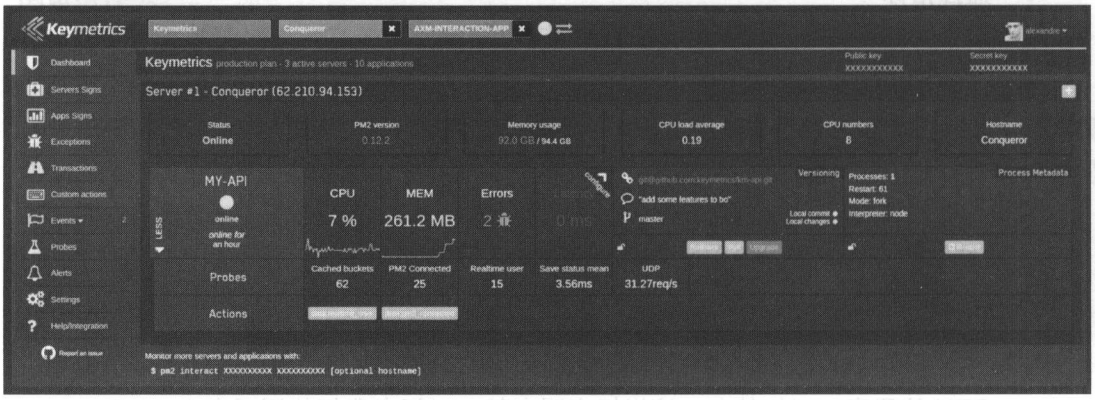
如果我们的服务器是 Linux 机器，查询硬件指标将会非常容易。在 Linux 上，所有硬件资源的“魔法”都发生在`/proc`文件夹里。该文件夹由系统内核维护，里面包含的文件记录了系统多个方面的表现状态。

软件指标则较难收集，我们将使用 PM2 作者的另一作品 Keymetrics 来监控 Node.js 应用。

采用 PM2 和 Keymetrics 进行监控

我们之前已经接触过 PM2，这是一个用于运行 Node 应用的强大装置，同时也能很好地在生产服务器上监控独立应用。然而，就我们的业务场景来说，通常并不容易访问到生产环境。

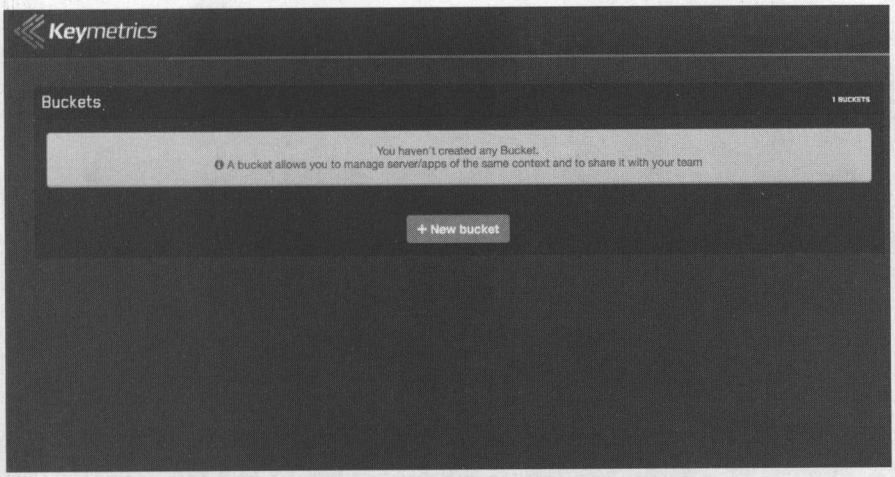
PM2 的作者通过创建 Keymetrics 解决了这一问题。Keymetrics 是一款软件即服务（SaaS）的组件，它允许 PM2 将监控数据通过网络发送给其站点，如下图所示（可以在`https://keymetrics.io/`访问到该页面）：



虽然 Keymetrics 并不是免费的，但是它提供了免费版来演示其工作方式。我们在本章

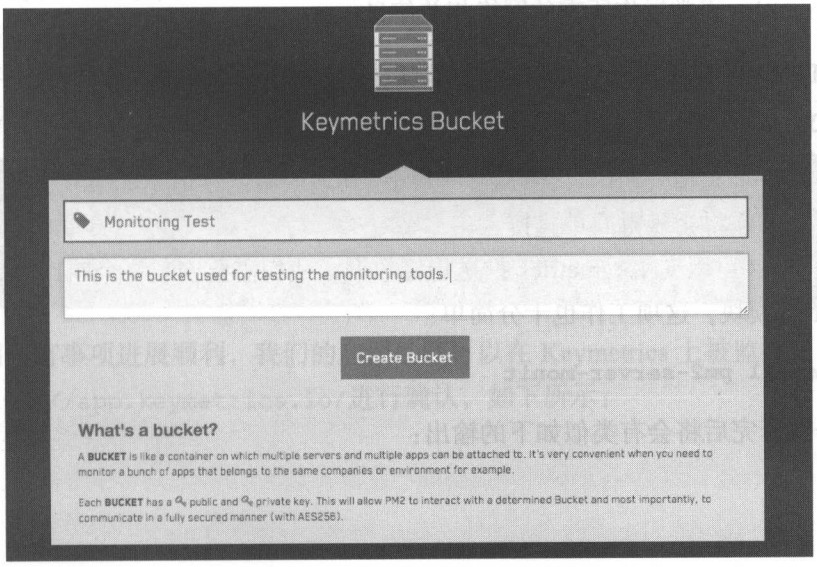
中将会使用免费版来进行讲解。

首先，需要注册一个用户账户。一旦我们登录新注册的账户，将会看到如下提示：

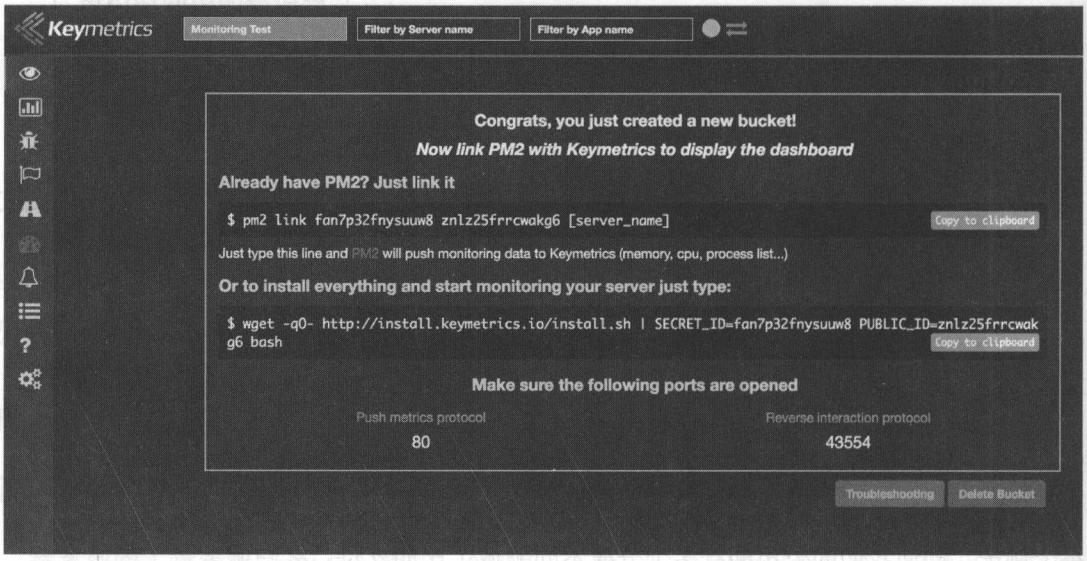


该页面提示我们去创建一个 bucket。Keymetrics 使用 bucket 这个概念来定义上下文。举个例子，即使我们的公司拥有多个分属不同领域（比如支付、客户服务等）的服务器，也可以在一个 bucket 中监控所有的服务器。对于一个 bucket 中可以容纳多少服务器并没有限制，甚至可以让整个公司共享一个 bucket，从而让访问变得非常简单。

让我们来创建一个名为 `Monitoring Test` 的 bucket，如下图所示：



非常简单，一旦单击 Create Bucket 按钮，Keymetrics 将会向我们展示需要启动应用监控的相关信息，如下图所示：



正如你所看到的，页面展示了 Keymetrics 所用私钥的相关信息。通常，我们不会选择将私钥共享给其他人。

正如页面上所示，下一步需要进行配置，将 PM2 的数据推送到 Keymetrics。同时，还有一些 Keymetrics 正常运作所需的网络相关信息：

- PM2 将会向 Keymetrics 所在的 80 端口推送数据。
- Keymetrics 将会向 43554 端口回传数据。

通常，在一个大型公司内，会存在一些网络环境的限制，但是如果我们在家里进行测试的话，所有事项将会显得简单直接。

为了让 PM2 可以向 Keymetrics 推送指标数据，我们需要安装一个叫作 `pm2-server-monit` 的 PM2 模块，这项工作也十分简单：

```
pm2 install pm2-server-monit
```

该命令执行完后将会有类似如下的输出：

```
[PM2][Module] Installing module pm2-server-monit
[PM2][Module] Processing...
..pm2-server-monit@1.1.0 .pm2/node_modules/pm2-server-monit
├── cpu-stats@1.0.0
├── shelljs@0.6.0
└── pmx@0.6.0 (json-stringify-safe@5.0.1, debug@2.2.0)
[PM2][Module] Module downloaded
[PM2] Process launched
[PM2][Module] Module successfully installed and launched
[PM2][Module] : To configure module do
[PM2][Module] : $ pm2 conf <key> <value>
┌──────────┬────┬──────┬─────┬────────┬─────────┬────────┬──────────┐
│ App name │ id │ mode │ pid │ status │ restart │ uptime │ memory │ watching │
└──────────┴────┴──────┴─────┴────────┴─────────┴────────┴──────────┘
Module activated
┌──────────────────┬─────────┬────────────┬────────┬─────────┬──────┬──────────┐
│ Module           │ version │ target PID │ status │ restart │ cpu  │ memory   │
├──────────────────┼─────────┼────────────┼────────┼─────────┼──────┼──────────┤
│ pm2-server-monit │ N/A     │ 2032       │ online │ 0       │ 0%   │ 4.547 MB │
└──────────────────┴─────────┴────────────┴────────┴─────────┴──────┴──────────┘
Use `pm2 show <id|name>` to get more details about an app
```

让我们来运行页面上提示的以下命令：

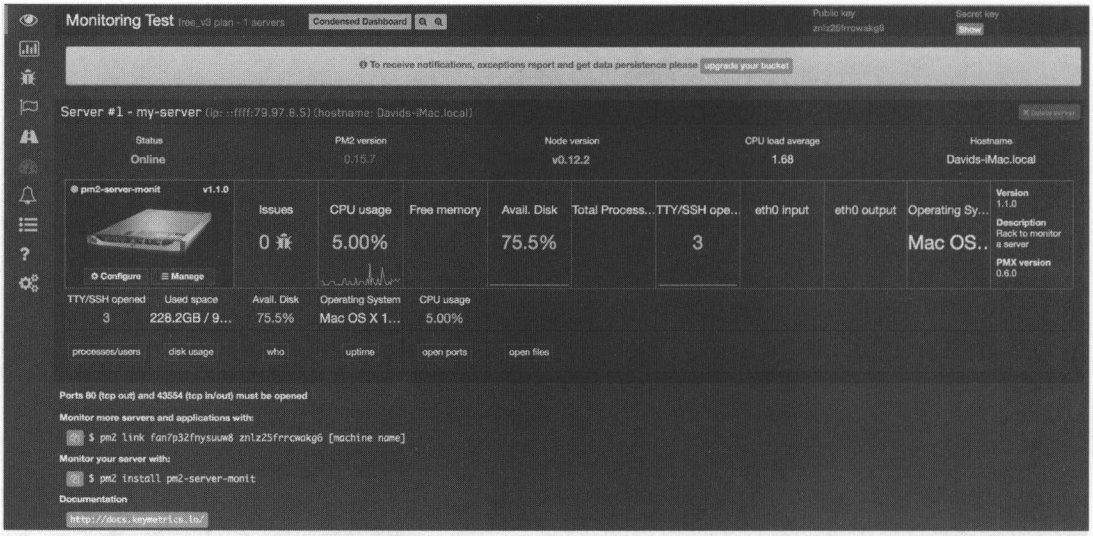
```
pm2 link fan7p32fnysuuw8 znlz25frrcwakg6 my-server
```

在这个例子中，我将[server name]替换成了my-server。对于服务器的名字是没有限制的；然而，如果要将 Keymetrics 应用于真实的系统，那么建议你为它选择一个具有描述性的名字从而能在页面的仪表盘上对服务器做出区分。

上一个命令将会产生如下输出：

```
[Keymetrics.io] Using (Public key: znlz25frrcwakg6) (Private key:
fan7p32fnysuuw8)
[Keymetrics.io] [Agent created] Agent ACTIVE - Web Access:
https://app.keymetrics.io/
```

这表明所有事项进展顺利，我们的应用已经可以在 Keymetrics 上被监控了，你可以通过访问 https://app.keymetrics.io/进行确认，如下所示：



现在，我们的服务器已经展示在了交互界面上。正如我们之前提到的，该 bucket 可以监控不同的服务器。我们已经创建了一个简单的虚拟机器，可以看一下页面的底部，Keymetrics 提供了可供执行的用于添加其他服务器的命令。在这个案例中，我们使用了 Keymetrics 的免费版本，所以仅能监控一台服务器。

一起来看看 Keymetrics 为我们提供了哪些功能。首先，可以看到例如像 CPU 使用率、可用内存和可用磁盘等指标。

通过所有这些硬件指标，我们可以了解系统当前的表现。尤其是在压力环境下，通过这些指标能很好地识别出哪些方面需要更多的硬件资源。

通常，硬件资源的使用情况也是应用故障的主要指示。现在，我们来看看如何使用 Keymetrics 来诊断问题。

问题诊断

由于软件缺陷而造成的内存泄露问题通常并不好解决。让我们来看看下面的代码。

先运行一段程序，该程序使用了一个简单的 `seneca.act()` action：

```
var seneca = require('seneca')();

var names = [];

seneca.add({cmd: 'memory-leak'}, function(args, done){
```

```
    names.push(args.name);
    greetings = "Hello " + args.name;
    done(null ,{result: greetings});
  });

  seneca.act({cmd: 'memory-leak', name: 'David'}, function(err,
    response) {
    console.log(response);
  });
```

该程序有着明显的内存泄露问题，当然这是我刻意为之的。names 数组将无限制地保持增长。在前面的例子中，由于我们的应用只是一个脚本，它在启动和结束时都无须在内存中保持状态，所以这并不是什么大问题。

请记住，在 JavaScript 中，如果在定义变量的时候没有使用 var 关键词，那么该变量是分配在全局范围内的。

一旦有人在应用的其他部分复用了我们的代码，问题就会暴露出来。

让我们假设一个场景，系统已经成长到了一个点，急需一个微服务来接待新用户（或者处理类似姓名、偏好及配置等个人信息的初始提交）。下面的代码演示了如何来构建该服务：

```
var seneca = require('seneca')();

var names = [];

seneca.add({cmd: 'memory-leak'}, function(args, done){
  names.push(args.name);
  greetings = "Hello " + args.name;
  done(null ,{result: greetings});
});

seneca.listen(null, {port: 8080});
```

在这个例子中，Seneca 将会监听来自 Seneca 客户端或其他类似 curl 这些系统的 HTTP 请求。当运行该应用时，将有如下输出：

```
node index.js

2016-02-14T13:30:26.748Z szwj2mazorea/1455456626740/40489/- INFO hello
Seneca/1.1.0/szwj2mazorea/1455456626740/40489/-
2016-02-14T13:30:27.003Z szwj2mazorea/1455456626740/40489/- INFO listen
{port:8080}
```

然后在另一个终端里，我们使用 curl 来作为微服务的客户端，然而虽然诸事平顺，但是内存泄露却正在悄然增长：

```
curl -d '{"cmd": "memory-leak", "name":"David"}' http://127.0.0.1:8080/act
{"result":"Hello David"}%
```

接着，我们将使用 Keymetrics 来定位问题。第一件事就是使用 PM2 来重新运行程序。为此，需要运行如下命令：

```
pm2 start index.js
```

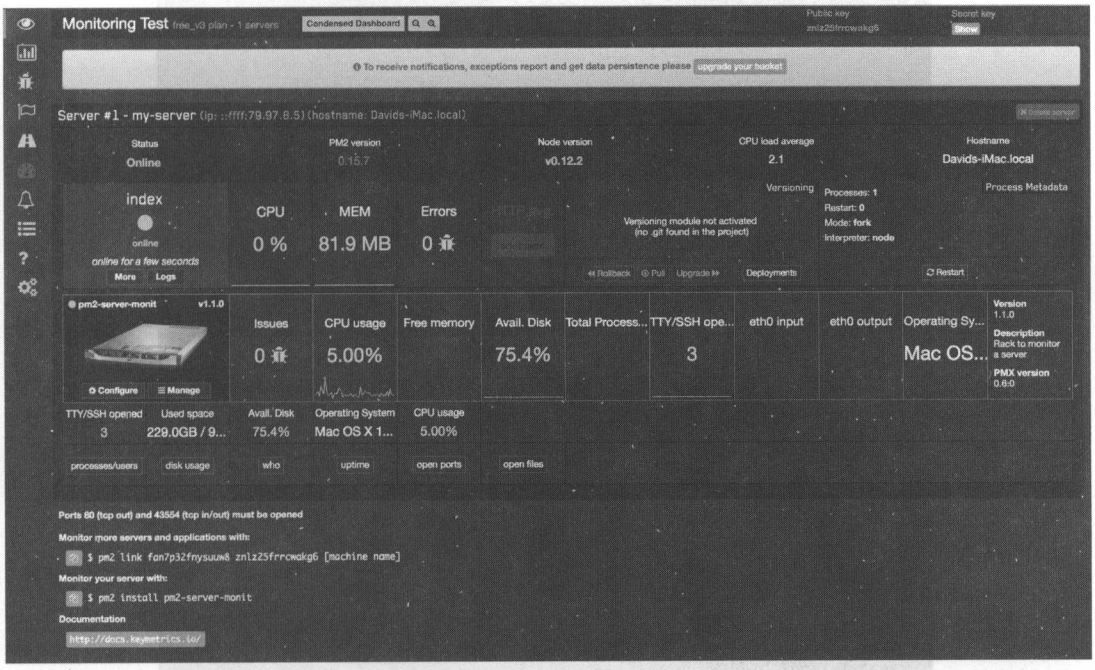
该命令将会产生如下输出：

```
→ code  pm2 start index.js
[PM2] Starting index.js in fork_mode (1 instance)
[PM2] Done.
• Agent online - public key: znlz25frrcwakg6 - machine name: my-server - Web access: https://app.keymetrics.io/
┌──────────┬────┬──────┬───────┬────────┬─────────┬────────┬──────────┐
│ App name │ id │ mode │ pid   │ status │ restart │ uptime │ memory   │ watching │
├──────────┼────┼──────┼───────┼────────┼─────────┼────────┼──────────┼──────────┤
│ index    │ 1  │ fork │ 41883 │ online │ 0       │ 0s     │ 8.348 MB │ disabled │
└──────────┴────┴──────┴───────┴────────┴─────────┴────────┴──────────┴──────────┘
Module activated
┌──────────────────┬─────────┬────────────┬────────┬─────────┬─────┬──────────┐
│ Module           │ version │ target PID │ status │ restart │ cpu │ memory   │
├──────────────────┼─────────┼────────────┼────────┼─────────┼─────┼──────────┤
│ pm2-server-monit │ 1.1.0   │ 7139       │ online │ 0       │ 0%  │ 81.930 MB │
└──────────────────┴─────────┴────────────┴────────┴─────────┴─────┴──────────┘
Use `pm2 show <id|name>` to get more details about an app
```

让我们来对该输出内容进行一下说明：

- 第一行给出了关于与 Keymetrics 集成的一些信息。例如公钥、服务器名以及访问 Keymetrics 的 URL 等。
- 在第一个表中，我们可以看到处于运行中的应用的名字，以及内存、运行时间和 CPU 等统计信息。
- 在第二个表中，我们可以看到与 `pm2-server-monit` PM2 模块相关的信息。

现在，让我们访问 Keymetrics，来看看发生了什么：



应用已在 Keymetrics 完成注册，我们可以在控制面板里看到该应用的相关信息。

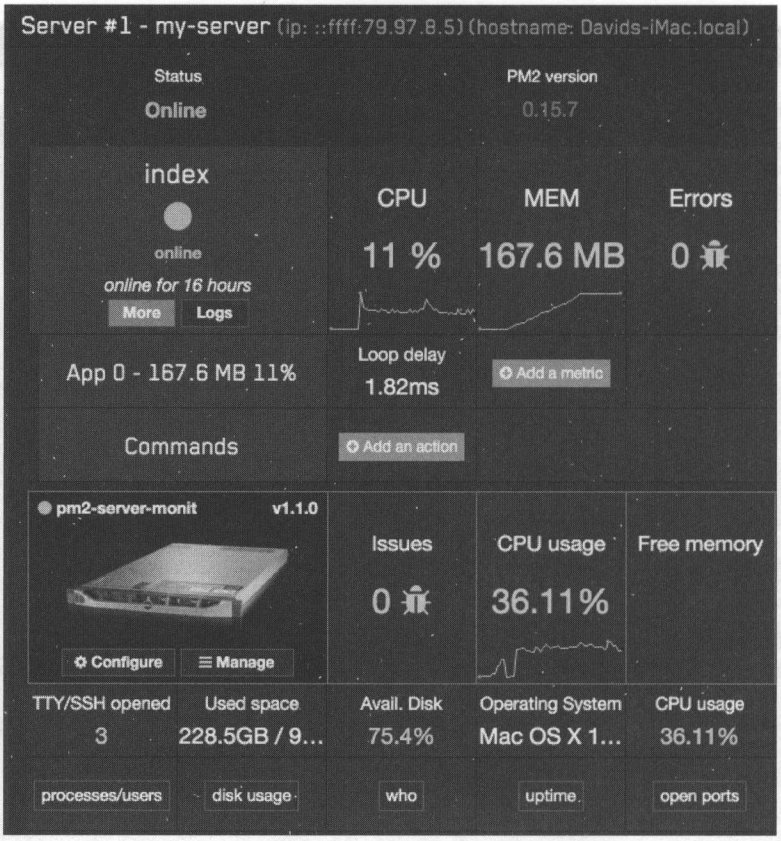
正如你所看到的，我们的应用已经展示在了 Keymetrics 上。

显而易见，我们可以看到应用的很多有用信息。其中便有内存的使用信息，该指标将会揭示出内存的泄露问题，因为它会保持持续增长。

现在，我们将在应用中强制造成一个由内存泄露引起的问题。在当前情况下，我们只需要启动服务器（即我们之前编写的 `index.js` 应用），然后向它发起大量的请求：

```
for i in {0..100000}
do
  curl -d '{"cmd": "memory-leak", "name":"David"}'
    http://127.0.0.1:8080/act
done
```

这只是一个简单的 bash 脚本，但已足够打开应用的潘多拉之盒：

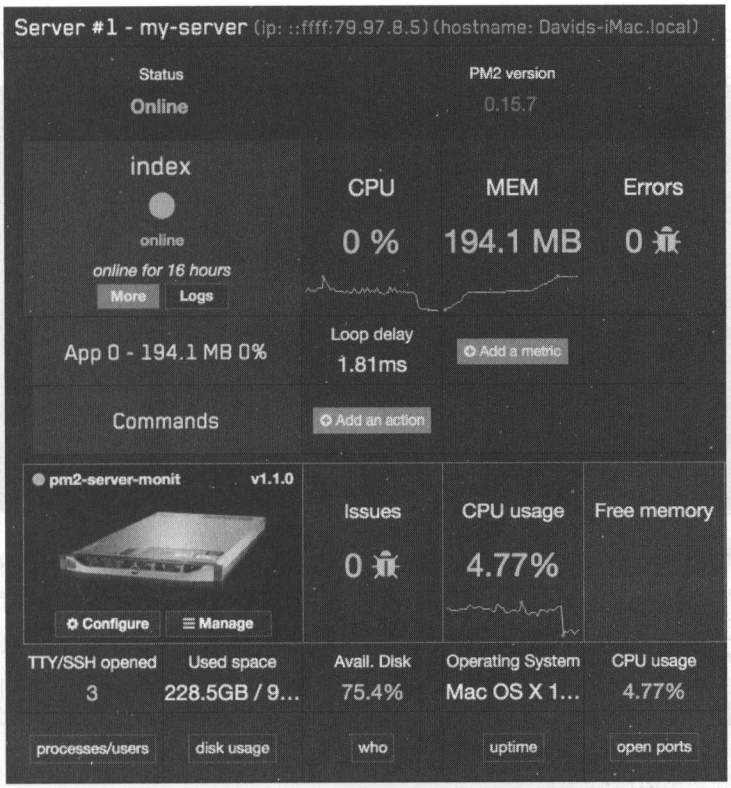


可以看到，当前应用的 CPU 负载已经很高（CPU 达到 36%的使用率，内存使用也增加到了 167MB）。

上图展示了循环请求对系统造成的影响。让我们来对此进行一下说明：

- 由于应用和 bash 脚本都使用了大量的资源，以至于应用的 CPU 使用率上升至 11%，而其平均循环延时为 1.82 毫秒。至于系统整体而言，总 CPU 使用率飙升至 36.11%。
- 内存消耗也从 81.9 MB 飙升至 167.6 MB。正如你所看到的，内存图中的曲线并没有保持直线上升，这是由于在进行垃圾回收。在 Node.js 中，垃圾回收是将不再被引用的对象从内存中释放掉的一项活动，这使得系统能重复利用硬件资源。
- 就错误而言，我们的应用非常稳定，错误数为 0（稍后我们将回来继续讨论这一部分）。

现在，一旦 bash 脚本运行结束（由于它过于耗费时间和资源，我是手动结束它的），我们便可以在下图中再次看到系统发生的情况：



可以看到 CPU 使用率恢复到了正常水平，那么内存呢？由于我们的程序存在内存泄露，所以只要内存被变量引用着（别忘了，`names` 数组在不断地累加着名字），应用消耗的内存自始至终都无法被释放。

在这个场景里，由于例子本身非常简单，所以可以很清楚地展示内存泄露的问题所在。但是在一些复杂的应用中，通常并不会这么明显。由于我们通常对应用新版本的部署较为频繁，会导致这个错误但可能永远不会作为问题暴露出来。

监控应用异常

应用错误通常是指应用无法处理意外情况时发生的一些事件，比如除数为零或尝试访问应用没有定义的属性，通常就会导致这些问题。

当你面对的是像 Tomcat 中所使用的这种多线程框架（或语言）时，某个线程如果因某个异常而死亡通常只会影响一个用户（运行在该线程上的用户）。但是在 Node.js 中，某个冒出来的异常可能会对我们的应用造成致命的伤害。

PM2 和 Seneca 能很好地保证应用的存活，因为 PM2 会在应用被意外停止时重启应用，而 Seneca 会在某个行为导致异常发生时防止应用死亡。

Keymetrics 开发了一个叫作 pmx 的模块，通过它我们能够以编程的方式在错误发生时得到通知：

```
var seneca = require('seneca')();

var pmx = require('pmx');

var names = [];

seneca.add({cmd: 'exception'}, function(args, done){
  pmx.notify(new Error("Unexpected Exception!"));

  done(null ,{result: 100/args.number});
});

seneca.listen({port: 8085});
```

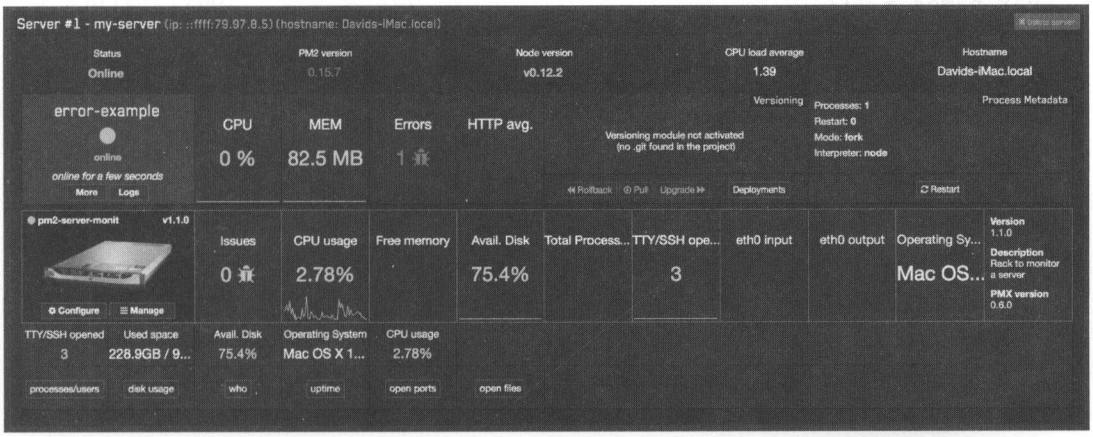
这段代码很简单且其意义不言自明：如果发送的参数 number 是 0，那么向 Keymetrics 发送一个异常。如果我们运行这段代码，将会产生如下输出：

```
→ code  pm2 start error-example.js
[PM2] Starting error-example.js in fork_mode (1 instance)
[PM2] Done.
• Agent online - public key: znlz25frrcwakg6 - machine name: my-server - Web access: https://app.keymetrics.io/
┌─────────────┬────┬──────┬───────┬────────┬─────────┬──────────┬──────────┐
│ App name    │ id │ mode │ pid   │ status │ restart │ uptime   │ memory   │ watching │
├─────────────┼────┼──────┼───────┼────────┼─────────┼──────────┼──────────┼──────────┤
│ error-example │ 7 │ fork │ 64724 │ online │ 0       │ 0s       │ 6.738 MB │ disabled │
└─────────────┴────┴──────┴───────┴────────┴─────────┴──────────┴──────────┴──────────┘
Module activated
┌──────────────┬─────────┬────────────┬────────┬─────────┬─────┬──────────┐
│ Module       │ version │ target PID │ status │ restart │ cpu │ memory   │
├──────────────┼─────────┼────────────┼────────┼─────────┼─────┼──────────┤
│ pm2-server-monit │ 1.1.0 │ 7139     │ online │ 0       │ 0%  │ 79.922 MB │
└──────────────┴─────────┴────────────┴────────┴─────────┴─────┴──────────┘
Use `pm2 show <id|name>` to get more details about an app
```

现在需要让服务器命中条件而触发一个错误。跟之前一样，我们会使用 curl 来进行测试：

```
curl -d '{"cmd": "exception", "number": "0"}' http://localhost:8085/act
{"result":null}%
```

现在，当我们访问 Keymetrics 时，可以看到已经有一个错误被记录下来，如下图所示：



Keymetrics 另一个有趣的地方是关于通知的配置。由于 PM2 会发送系统中的几乎每一个指标数据，而我们则可以出于应用健康度的考虑来配置 Keymetrics 的阈值。

这有助于我们将通知集成到公司的聊天工具（类似 Slack 这样的协同办公聊天工具）中，并在应用出现问题时实时得到告警。

自定义指标

Keymetrics 同样也允许我们使用探针（probe）。探针是一个自定义指标，应用可以编程的方式将它发送到 Keymetrics。

在 Keymetrics 的原生库中，有很多不同类型的值可以供我们直接进行推送。下面来看看最有用的几个。

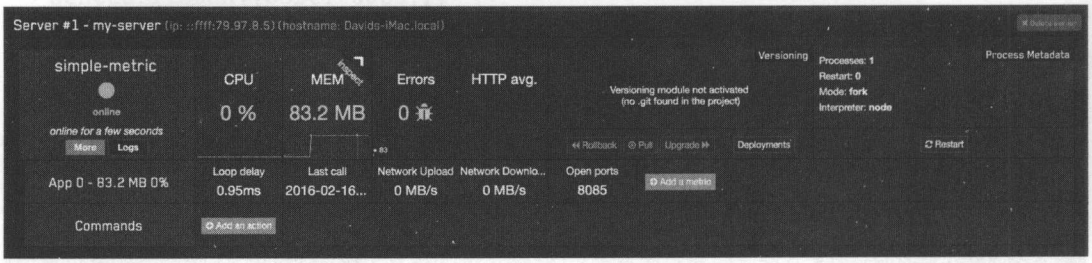
简单指标

正如其名字所表明的，简单指标是非常基础的指标，开发者可以为将要发送到 Keymetrics 的数据设置值：

```
var seneca = require('seneca')();
var pmx = require("pmx").init({
  http: true,
  errors: true,
  custom_probes: true,
  network: true,
  ports: true
```

```
});
var names = [];
var probe = pmx.probe();

var meter = probe.metric({
  name     : 'Last call'
});

seneca.add({cmd: 'last-call'}, function(args, done){
  console.log(meter);
  meter.set(new Date().toISOString());
  done(null, {result: "done!"});
});

seneca.listen({port: 8085});
```

在该例子中，该指标会在该 action 被调用的时候向 Keymetrics 发送当前的时间：



而该指标几乎是零配置的：

```
var probe = pmx.probe();

var meter = probe.metric({
  name     : 'Last call'
});
```

在该指标中几乎没有复杂性。

计数器

这个指标可用于计算某个事件的发生次数：

```
var seneca = require('seneca')();
```

```
var pmx = require("pmx").init({
  http: true,
  errors: true,
  custom_probes: true,
  network: true,
  ports: true
});
var names = [];
var probe = pmx.probe();

var counter = probe.counter({
  name : 'Number of calls'
});

seneca.add({cmd: 'counter'}, function(args, done){
  counter.inc();
  done(null, {result: "done!"});
});

seneca.listen({port: 8085});
```

在前面的代码中，我们可以看到计数器会在“counter”action 每次被访问时得到累加。该指标也允许我们通过调用计数器的 dec() 方法来对值进行递减：

```
counter.dec();
```

平均计算值

该指标允许我们在某个事件发生时进行记录，并且会自动计算单位时间内的事件次数。这对计算平均值是相当有用的，同时也是衡量系统负载的绝佳工具。我们来看一个简单的例子，如下所示：

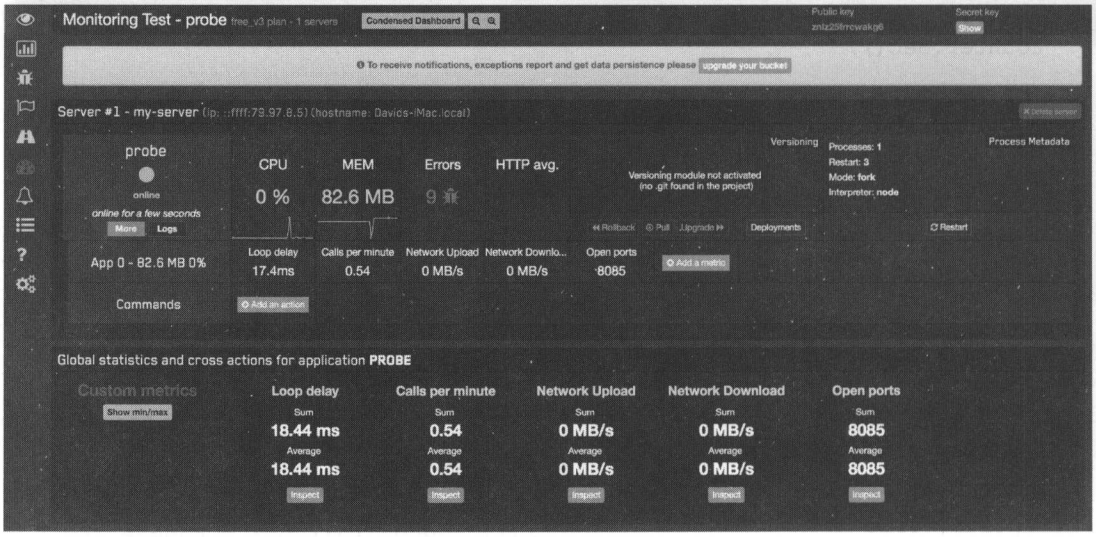
```
var seneca = require('seneca')();
var pmx = require("pmx").init({
  http: true,
  errors: true,
  custom_probes: true,
  network: true,
  ports: true
```

```
});
var names = [];
var probe = pmx.probe();

var meter = probe.meter({
  name       : 'Calls per minute',
  samples    : 60,
  timeframe  : 3600
});

seneca.add({cmd: 'calls-minute'}, function(args, done){
  meter.mark();
  done(null, {result: "done!"});
});

seneca.listen({port: 8085});
```

上述代码创建了一个探针，并且向 Keymetrics 发送了一个叫作 `Calls per minute`（每分钟的调用次数）的指标。

现在，如果我们在运行该应用的同时多次执行下面的命令，该指标就会展示在 Keymetrics 的界面上：

```
curl -d '{"cmd": "calls-minute"}' http://localhost:8085/act
```



正如你所看到的，用户界面上出现了一个叫作 Calls per minute 的指标。配置该指标的关键是下面这段初始化的代码：

```
var meter = probe.meter({
  name      : 'Calls per minute',
  samples   : 60,
  timeframe : 3600
});
```

你可以看到，在配置项字典里设置了该指标的名字，同时还有其他两个参数：samples 和 timeframe。

参数 samples 与计算指标的时间间隔有关，在这个例子中，由于统计的是每分钟的调用次数，所以时间间隔就是 60 秒。

参数 timeframe 表示我们希望 Keymetrics 将数据保持多久，简单地说就是指标分析所横跨的时间段。

类人猿大军——来自 Netflix 的主动监控

当我们在构建面向微服务的应用时，Netflix 是经常被作为参考对象的公司之一。这些公司在将实践经验归纳成新的思想方面显得极具创造性且才华横溢。

其中一个我特别喜爱的思想被它们称为“类人猿大军（Simian Army）”，而我则喜欢称它为*主动监控*。

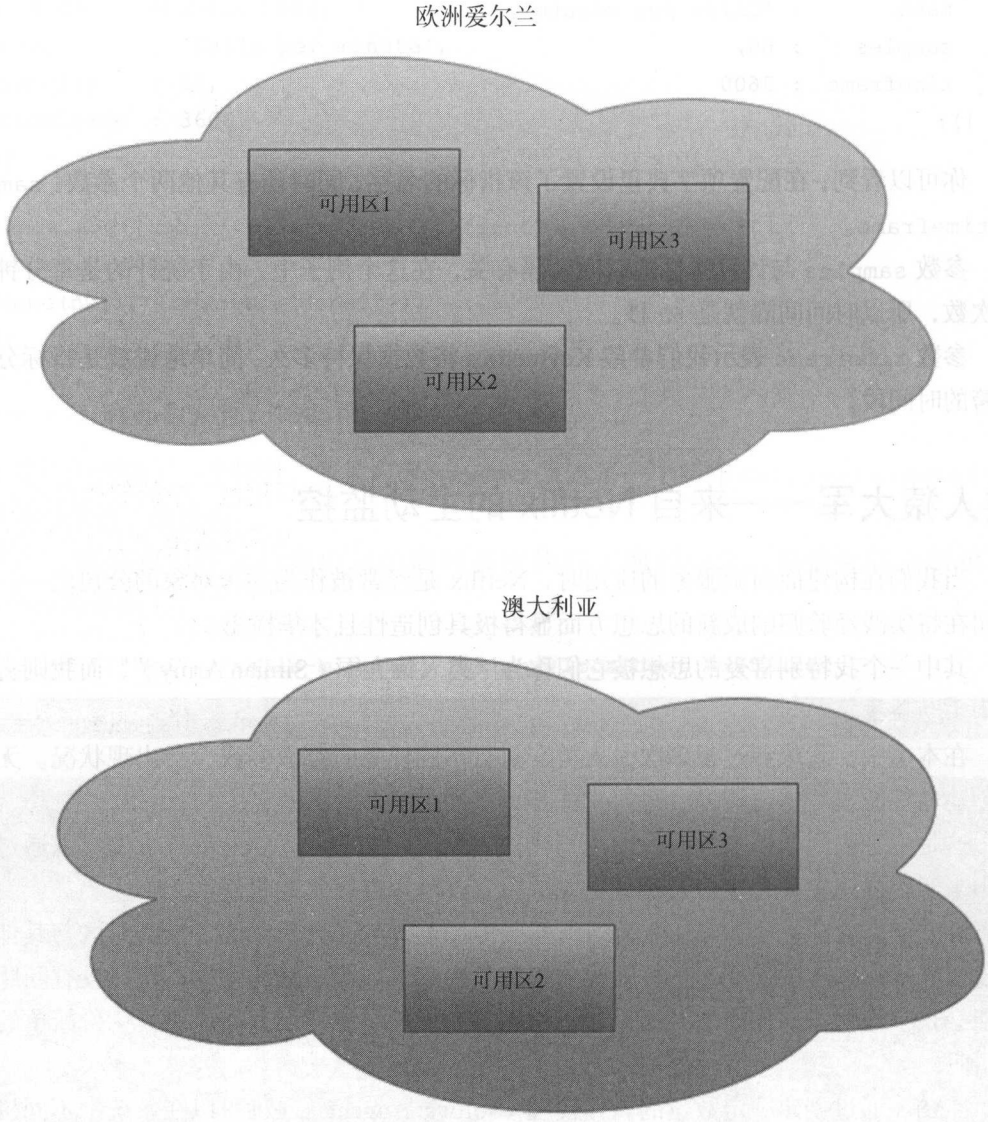
在本书中，我谈到过很多次，人类在执行不同任务时都或多或少会出现状况。无论你在创建软件的过程中花费多少努力，总还是会存在损害系统稳定性的 bug。

这是一个大问题，当我们在使用当代云提供商的服务时，你的基础设施可以通过一段脚本来实现自动化，而这个问题也变成了一个值得讨论的大型话题。

请想一下，在一个拥有数千台服务器的集群里，如果有三台服务器的时区与其他服务器没有同步的话会发生什么？好吧，这其实取决于你系统本身的特性，可能没有问题，也可能会是个大问题。你能想象银行在它的事务出现混乱的时候只向你出具一份简单的声明了事吗？

Netflix 通过创建一定数量的软件代理（software agent，一种可以在系统中不同机器间移动的程序）解决（或缓解）了前面提到的问题。为了确保基础设施的健壮性，我们会根据不同的目的将这些代理命名成不同种类猴子的名称。

如果你曾使用过 Amazon Web Services，那么你可能已经知道，机器和计算元素会被划分成不同的“区域（region）”（比如 EU、Australia、USA 等）。在每个区域内，又会有多个“可用区（availability zones）”，如下图所示：



这让作为工程师的我们可以创建出没有单点故障的软件和基础设施。

单点故障是指单个元素的故障将会导致整个系统表现异常的情况。

这样的结构又给 Netflix 的工程师们提出了另一些问题，如下所示：

- 如果我们盲目信任自己的设计而没有通过测试来确定是否确实有某些故障点会发生什么？
- 如果整个“可用区”或“区域”都宕机了会发生什么？
- 如果由于某些原因出现反常的延迟，我们的应用会如何表现？

为了解答这些问题，Netflix 创建了多种代理。代理是一个运行在系统（在这个例子中就是我们的微服务系统）上的软件，它可以执行多种操作，例如检查硬件、检测网络延迟等。代理的思想并不是新提出来的，但是直到现在，对其的应用仍近乎是一个未来话题。让我们来看看由 Netflix 创建的这些代理。

- **Chaos Monkey（混乱猴子）**：该代理会断开指定“可用区”中的健康机器的网络。这可以确保在“可用区”内没有单点故障。因此，如果我们的应用均衡运行在 4 个节点上的话，一旦 Chaos Monkey 闯入，它将断开 4 台中的 1 台机器。
- **Chaos Gorilla（混乱大猩猩）**：它与 Chaos Monkey 非常相似，Chaos Gorilla 会断开整个“可用区”，从而验证 Netflix 服务会调整到其他“可用区”上。换句话说，Chaos Gorilla 是 Chaos Monkey 的大哥；其中 Chaos Monkey 操作的是服务器级别，而 Chaos Gorilla 操作的是分区级别。
- **Latency Monkey（延迟猴子）**：该代理负责在连接中引入“人为的”延迟。对于系统开发来说，通常很少考虑延迟的问题，但是在构建微服务的架构时，延迟却是一个微妙的话题：一个节点的延迟可能会影响整个系统的服务质量。当一个服务耗尽资源时，通常第一个征兆就是响应的延迟；因此，Latency Monkey 是了解系统在压力下表现行为的好办法。
- **Doctor Monkey（医生猴子）**：健康检查就是指访问应用的端点，如果万事皆顺则返回 HTTP 200，如果应用中存在任何问题则返回 500 错误码。Doctor Monkey 是一个可以随机对应用中的节点执行健康检查的代理，它会报告出错的节点，从而可以对这些节点进行替换。

- **10–18 Monkey（本地及国际化猴子）**：像 Netflix 这样的公司是全球性的，所以它们需要具备语言识别能力（当然，你不会希望以西班牙语为母语的老母亲，在访问网站时看到的却是德语）。10-18 Monkey 会对那些配置错误的实例进行报告。

还有很多其他类型的代理，但是我只想向你说明清楚什么是主动监控。当然，这类监控对于小公司来说是力所不逮的，但是通过了解这些监控类型的存在，我们可以从中得到启发并在构建自有监控装置的时候受益。

Simian Army 的代码基于 Apache 许可证发布，并托管在以下仓库：`https://github.com/Netflix/SimianArmy`。

一般而言，主动监控是符合“尽早失败”这一哲学的。在这方面，我是个老手。不管系统的问题有多大或多严重，你都想尽早地发现它，理想情况下是不要影响到任何用户。

吞吐量和性能降级

吞吐量对我们的应用而言，相对于工厂的月产量。它是衡量应用性能表现的指标，同时也可以解答很多系统的问题。

与吞吐量非常接近的另一个可以衡量的指标是：延迟。延迟是指执行完工作需要多久的性能单元。

让我们来看看下面这个例子：

我们的应用是基于微服务架构的，它负责计算抵押贷款申请人的信用评级。由于拥有大量的客户（这是一个好问题所需具备的条件），我们决定以批量的方式来处理这些申请。让我们来针对这个问题构思一个简短的算法：

```
var seneca = require('seneca')();
var senecaPendingApplications = require('seneca').client({type:
  'tcp',
  port: 8002,
  host: "192.168.1.2"});
  var senecaCreditRatingCalculator =
    require('seneca').client({type: 'tcp',
  port: 8002,
  host: "192.168.1.3"});
```

```
seneca.add({cmd: 'mortgages', action: 'calculate'}, function(args, callback) {
  senecaPendingApplications.act({
    cmd: 'applications',
    section: 'mortgages',
    custmers: args.customers}, function(err, responseApplications) {
      senecaCreditRatingCalculator.act({cmd: 'rating',
        customers: args.customers}, function(err, response) {

        processApplications(response.ratings,
        responseApplications.applications,
        args.customers
      );
    });
  });
});
```

这是一个简短的 Seneca 应用（只是理论上的演示，请勿尝试运行该程序，因为其中省略了很多代码），它扮演的是其他两个微服务的客户端，如下所示：

- 第一个微服务可用于获取抵押贷款申请的待处理列表。
- 第二个微服务可获取请求客户的信用等级。

这是一个处理抵押贷款申请的真实场景。平心而论，我之前曾参与过一个与之非常相似的系统，虽然之前的系统更加复杂，但是其工作流与这个场景非常相似。

让我们来继续讨论吞吐量和延迟。假设我们拥有一大批抵押贷款需要处理，而此时系统已经表现异常：网络已经不再像之前那样快速，甚至开始出现丢包。

一些申请开始被丢弃，并且需要进行重试。在理想情况下，我们的系统可以提供每小时处理 500 个申请的吞吐量，平均每个申请的处理需要耗时 7.2 秒。然而，正如我们刚才提到的，系统的表现状态并不佳；每小时只能处理 270 个申请，其中每个抵押贷款的申请需要耗时 13.3 秒。

正如你所看到的，根据延迟和吞吐量，我们可以通过之前的经验来衡量业务的事务处理表现：当前的处理能力只达到了正常能力的 54%。

这将会是一个严重的问题。说实在的，像这样的下降应该马上打开系统中的所有告警，因为我们的基础设施可能确实已经出现严重的问题；然而，如果我们将系统开发得足够智能，那么只会发生性能的降级，而系统则不会停止工作。可以利用断路器和像 RabbitMQ

这样的队列技术来实现这一目标。

队列是一个展示了如何在 IT 系统中借鉴人类行为的不错的例子。非常重要的是，通过将服务生产或消费的一条简单的消息作为连接点，从而简单地将软件组件解耦这一技巧，可以在编写复杂软件时给我们带来很大的优势。

在 HTTP 上使用队列的另一个优势是，一旦网络出现丢包，HTTP 的消息会被直接丢弃。

我们希望构建的应用能实现要么完全成功，要么完全出错。通过使用 RabbitMQ 这样的队列技术，可以异步传递消息，无须担心间歇性的故障：一旦将消息传递到相应的队列，将会立即将消息进行持久化，等待直到客户端可以消费它（或者直到消息超时）。

这使我们可以在基础设施层面处理好间歇性错误，从而基于围绕队列的通信构建更加健壮的应用。

不得不说，Seneca 让我们的生活变得更加轻松：在 Seneca 框架的插件系统中编写传输插件是一件相当简单的工作。

可以在下面的 GitHub 仓库找到 Seneca 的 RabbitMQ 传输插件：

`https://github.com/senecajs/seneca-rabbitmq-transport`

当然也还有很多其他的传输插件可供选择，而我们自己也可以根据需求来创建新的（或者修改现有的）插件。

如果你快速地瞥一眼 RabbitMQ 插件（只是举个例子），会发现我们需要做的就是编写一个传输插件来覆盖下面两个 Seneca 动作：

- `seneca.add({role: 'transport', hook: 'listen', type: 'rabbitmq'}, ...)`
- `seneca.add({role: 'transport', hook: 'client', type: 'rabbitmq'}, ...)`

通过使用队列技术，我们的系统将会在间歇性故障面前显得更加健壮，我们可以通过对性能降级来避免由于消息丢失而产生的灾难性故障。

小结

在本章中，我们深入探讨了如何通过 Keymetrics 来进行 PM2 监控。同时也学习了如何把监控做到位，从而能尽快发现应用的故障。

在我看来，保证质量是软件开发生命周期中最为重要的环节之一：不管你的代码看上

去多么棒，如果它不能正常工作，那都是毫无价值的。然而，如果要我选择另一个值得工程师们需要更加重视的环节，那我会选择部署。尤其需要更加明确的是，在每次部署中都要做好监控。只有这样，你才能在第一时间接到错误报告，才能足够快地做出响应来避免出现像数据错误或客户抱怨这样的重大问题。

同时我们也看到了由 Netflix 带来的关于主动监控的例子，这部分的执行或许并不在你的公司的能力范围内，但是它会为你做好软件监控实践带来新的启发。

Keymetrics 只是作为适合采用 Node.js 来进行讲解的一个例子，因为它能很好地与 PM2 进行集成，但是可供你选择的优秀的监控系统可不止这一个，例如还有 AppDynamics。如果你想要监控的是一个内部系统，那么可以选择 Nagios。问题的关键是你需要弄清楚想要监控应用的哪些方面，然后再去寻找合适的监控服务供应商。

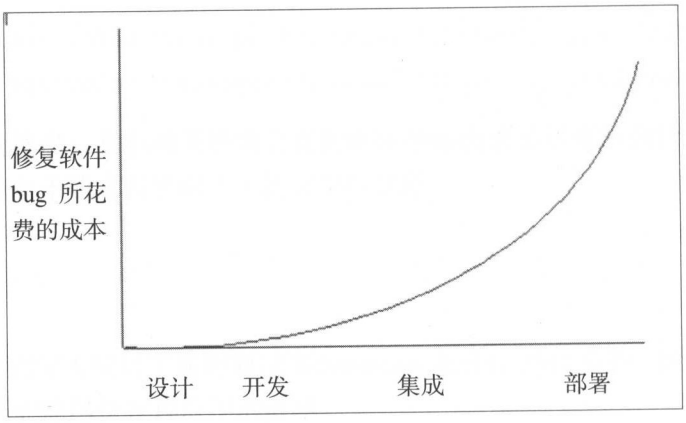
对于监控 Node.js 应用来说，还有两个很好的选择，分别是 StrongLoop 和 New Relic。它们跟 Keymetrics 是同一层面的，但是它们在面对大规模系统的时候会表现得更好，尤其是 StrongLoop，它是专门面向由 Node.js 编写的应用及微服务的。

8
微服务的部署

在这一章中，我们将开始部署微服务。我们将会采用多种不同的技术，帮助读者掌握必备的知识来为各项工作选择合适的工具。首先，我们会利用 PM2 及其部署能力，在远程服务器上将应用运行起来。接着，我们将会涉猎一下 Docker 及其围绕容器的整个生态系统，它是最为先进的部署平台之一。在这一章中，我们将会展示如何进行高度全自动化的部署。

软件部署的一些概念

部署通常是软件开发生命周期（SDLC）中最杂乱的一部分。在部署与系统管理之间总是缺了一环，而 DevOps 将在未来的几年内解决这些问题。在软件开发生命周期不同阶段修复 bug 所花费的成本如下图所示：



“尽早失败”是精益方法论中我最喜欢的一个概念。在变更管理的世界中，在软件开发生命周期不同阶段中修复一个 bug 所耗费的成本所形成的轨迹被称为“变更成本曲线”。

粗略估计，在生产环境修复一个 bug 的成本差不多是在需求讨论阶段的 150 倍。

不管这张图代表了什么，也不管所依据的方法论或使用的技术是什么，我们需要认识到的就是越早发现 bug 越节省时间。

从持续集成到持续交付，都应该尽可能地让流程自动化，而且是 100%的自动化。

记住一点，人类是不完美的，在手动执行重复性工作时容易出现错误。

持续集成

持续集成（CI）是每天（也可能不止一天）将不同分支的代码进行集成的一项实践，它通过运行集成测试和单元测试来验证变更不会破坏现有功能。

CI 应该自动进行，并使用和后续预生产及真实生产环境相同的基础设施配置，如此才能尽早地发现存在的问题。

持续交付

持续交付（CD）是一种软件工程方法，它的目标是构建体形小、可测试且易于部署的功能部件，以便于在任意时间进行无缝交付。

这也是我们构建微服务的目标。再次强调，应该推动交付流程的自动化，如果手动地去做这些事，那纯粹就是自找麻烦。

从微服务的角度来说，部署的自动化才是关键。我们需要承担拥有一堆服务（而不再是一堆机器了）的开销。如果没有做好自动化，我们将一直深陷“服务云”的维护，更别谈能为公司带来更多价值。

Docker 是我们的最佳拍档。有了 Docker 之后，为我们减少了许多部署新软件的麻烦，几乎只需在多个不同的环境之间移动文件（或容器）即可完成部署。我们将在本章的后续部分讨论这部分内容。

采用 PM2 进行部署

PM2 是一个极其强大的工具。不论我们处于开发中的哪个阶段，PM2 总可以为我们提供帮助。

而在软件开发的部署阶段，PM2 更是大放异彩。只需通过一个 JSON 配置文件，PM2

便可以管理整个应用集群，从而可以方便地在远程服务器上完成部署、重部署以及对应用的管理。

PM2 中的“生态系统”

PM2 将一组应用称为“生态系统（ecosystem）”。每一个“生态系统”都由一个 JSON 文件来描述，而生成该文件的最简单的方式就是运行下面这条命令：

```
pm2 ecosystem
```

该命令将会产生如下输出：

```
[PM2] Spawning PM2 daemon
[PM2] PM2 Successfully daemonized
File /path/to/your/app/ecosystem.json generated
```

根据 PM2 的版本不同，文件 `ecosystem.json` 的内容也会有所不同，下面的文件包含了一个 PM2 集群的骨架：

```
{
  apps : [

    {
      name      : "My Application",
      script    : "app.js"
    },

    {
      name      : "Test Web Server",
      script    : "proxy-server.js"
    }
  ],

*/
  deploy : {
    production : {
      user : "admin",
      host : "10.0.0.1",
```

```
      ref  : "remotes/origin/master",
      repo : "git@github.com:the-repository.git",
      path : "/apps/repository",
      "post-deploy" : "pm2 startOrRestart ecosystem.json --env
        production"
    },
    dev : {
      user : "devadmin",
      host : "10.0.0.1",
      ref  : "remotes/origin/master",
      repo : "git@github.com:the-repository.git",
      path : "/home/david/development/test-app/",
      "post-deploy" : "pm2 startOrRestart ecosystem.json --env
        dev",
    }
  }
}
```

该文件包含了分别根据两套环境配置的两个应用。我们将根据自身的需求来修改这个骨架，并为之前在第 4 章中编写的内容构建出整个“生态系统”。

尽管如此，让我们先来对该配置做一些解释：

- 我们拥有一个应用的数组，并定义了两个应用：API 和 WEB。
- 正如你所看到的，我们为每个应用都定义了一些配置参数。
  - `name`：应用的名称
  - `script`：应用的启动脚本
  - `env`：由 PM2 注入系统的环境变量
  - `env_<environment>`：与 `env` 相似，只是每一项都是为单个环境定制的
- 在默认的“生态系统”中，在 `deploy` 键的配置下定义了两套环境，如下所示：
  - `production`
  - `dev`

不难发现，两套环境的配置并没有非常大的差别，只是有一套环境被我们配置成了开发环境，同时还配置了各自用于部署应用的文件夹路径。

采用 PM2 来部署微服务

在第 4 章中，我们编写了一个电商系统来展示微服务中的不同概念和共同的关注点。现在，我们将去学习如何采用 PM2 来部署这些系统。

配置服务器

为了能采用 PM2 部署软件，我们首先需要对远程服务器进行配置，从而让本地机器可以通过公私钥机制来使用 SSH 与远程服务器对话。

完成这一工作非常容易，如下所示：

- 生成一个 RSA 的密钥
- 将该密钥安装到远程服务器上

让我们动手来完成它：

```
ssh-keygen -t rsa
```

该命令将会产生如下输出：

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/youruser/.ssh/id_rsa):
/Users/youruser/.ssh/pm2_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in pm2_rsa.
Your public key has been saved in pm2_rsa.pub.
The key fingerprint is:
eb:bc:24:fe:23:b2:6e:2d:58:e4:5f:ab:7b:b7:ee:38 dgonzalez@yourmachine.local
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|                 |
|                 |
|    .            |
|   o   S         |
|    o  ..        |
|   o o..o.       |
```

```
|  . +.+=E..      |
|   oo++**B+.     |
+-----------------+
```

现在，如果我们进入上述输出信息提到的目录，将可以看到以下两个文件。

- pm2_rsa：第一个文件是 pm2_rsa，这是你的私钥。正如“私”所体现的，不能让任何人访问这个私钥，否则他们可以在信任该私钥的服务器上盗用你的身份。
- pm2_rsa.pub：pm2_rsa.pub 是你的公钥。可以将该密钥发送给任何使用非对称加密技术的人，他们可以验证你的身份（或你愿意转借身份的人）。

接下来要做的就是将公钥复制到远程服务器上，这样一来，每当本地机器使用 PM2 尝试与服务器交谈时，它便会知道我们是谁，从而无须密码就可以登录 shell：

```
cat pm2_rsa.pub | ssh youruser@yourremoteserver 'cat >> .ssh/authorized_
keys'
```

最后一步就是在你的本地机器上将私钥注册为已知身份：

```
ssh-add pm2_rsa
```

仅此而已。

到此为止，无论何时使用用户名“youruser”SSH 到远程服务器上，你都无须再输入密码便可进入 shell：

一旦完成这些配置，就差最后一步便可以将任意应用部署到该服务器了：

```
pm2 deploy ecosystem.json production setup
pm2 deploy ecosystem.json production
```

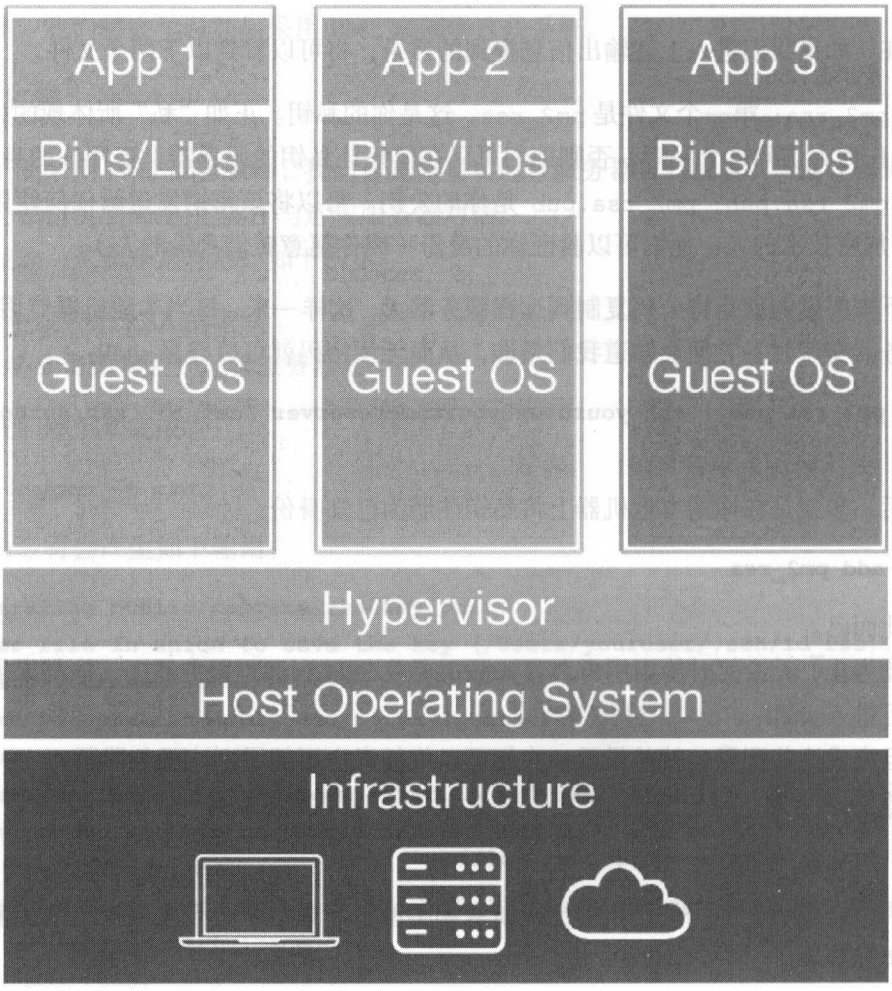
第一个命令将会根据应用所需完成所有相关配置。第二个命令会根据之前的配置完成应用的实际部署。

Docker——一种可用于软件交付的容器

虚拟化是近些年的一个大趋势。虚拟化使工程师们可以跨多个软件实例共享硬件。Docker 并不是真正意义上的虚拟化软件，但是从概念上来说是相似的。

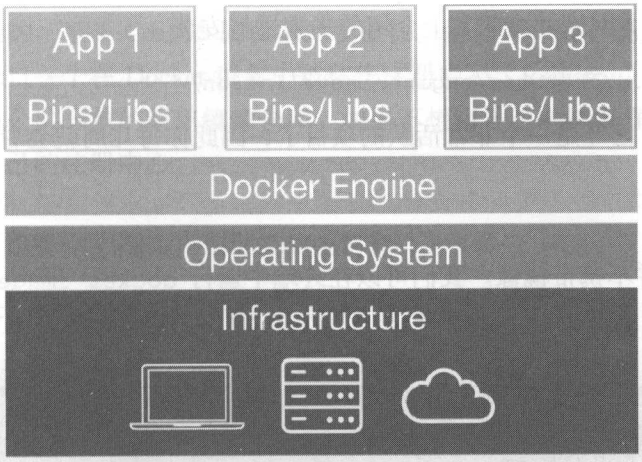
在一个纯粹的虚拟化方案中，新的操作系统运行在某个既有操作系统（宿主操作系统）的管理程序（hypervisor）之上。运行整个新操作系统意味着会消耗掉几个 GB 的硬盘，因

为需要对从内核到文件系统的全部内容进行复制，这样做非常消耗资源。这种虚拟化方案的结构如下图所示：



虚拟机环境的层次图

通过使用 Docker，我们只需要复制文件系统和一些二进制文件，因为我们并不需要运行整个操作系统。Docker 镜像通常只有几百 MB，而不像整个操作系统那样达到 GB 数量级的容量。它们相当轻量级，因此，我们可以在同一台机器上运行多个容器。上述使用 Docker 的结构如下图所示：



Docker的层次图

采用 Docker 之后，我们省去了一个软件部署环节的大麻烦，即配置管理。

我们从一个极其复杂并需要逐环境进行配置管理的境地进化到了容器时代，身处前者时需要担心如何对应用进行部署和配置，而在后者，目标环境只要是任意部署了 Docker 的机器即可，部署应用基本上就变得像安装软件包一样简单。

由于 Docker 需要利用大量高级的内核特性，如今仅有的原生支持 Docker 的操作系统只有 Linux。而 Windows 和 Mac 用户如果需要使用 Docker，则不得不运行一个 Linux 的虚拟机来为 Docker 容器提供支持。

组装容器

Docker 给我们提供了一种非常强大且熟悉（对开发者而言）的方式来配置容器。

你可以基于现有的镜像来创建容器（互联网上有着数以千计的镜像），也可以通过像添加新的软件包或修改文件系统这样的方式来修改已有的镜像，从而满足自己的需求。

一旦修改镜像满足需求之后，我们便可以使用一种类似于 Git 的版本控制系统来为新版的镜像创建容器。

不管怎样，我们首先要了解 Docker 是如何工作的。

安装 Docker

前面曾提到过，Docker 在 Mac 和 Windows 上是需要虚拟机来提供支持的。因此，在这些系统上的安装流程会比较特殊。想要了解如何才能在你的系统上以最佳的方式安装

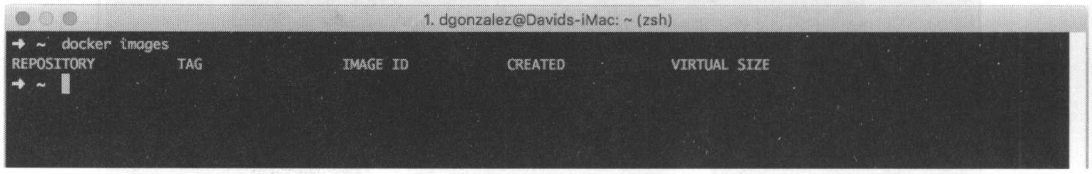
Docker，请移步官方网站并按照其上介绍的步骤进行安装：

```
https://docs.docker.com/engine/installation/
```

当前，Docker 已经是一个非常活跃的项目了，因此你每几周就会看到它有新的变化。

选择镜像

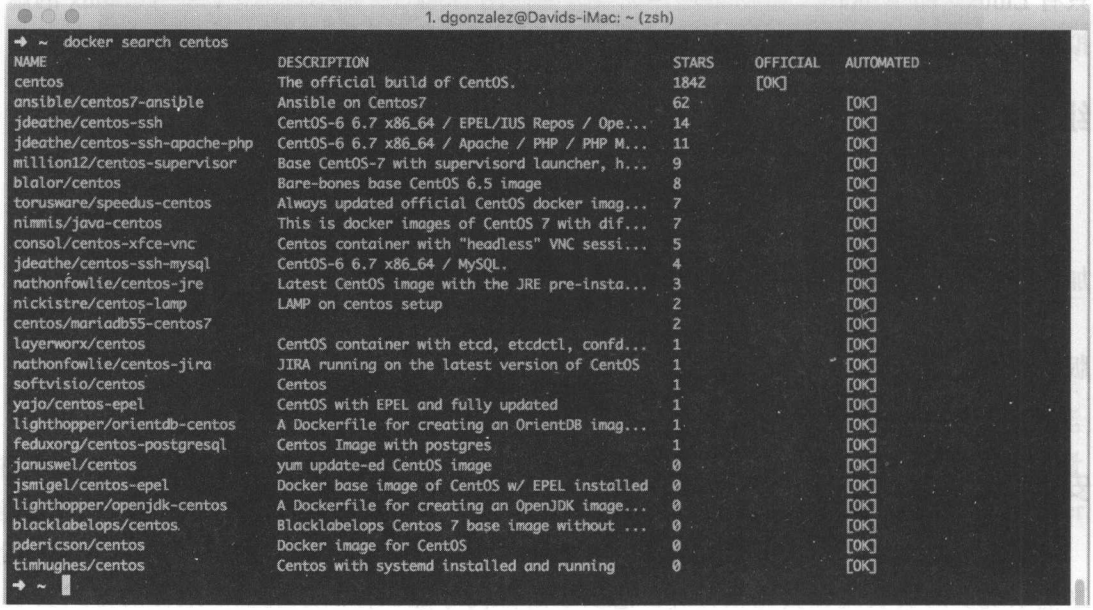
Docker 默认并不携带镜像。我们可以在终端中运行 `docker images` 来查看镜像，其输出如下所示：



可以看到目前是一个空的列表，本地机器上并没有任何镜像。我们需要做的第一件事就是查找镜像。在本例中，我们打算基于 CentOS 来创建镜像。CentOS 非常接近于 Red Hat Enterprise Linux，同时也是 Linux 目前在业界使用最为广泛的一个发行版。

业界为 CentOS 提供了大量的支持，且互联网上有着丰富的可用于故障定位的信息。

让我们来搜索一下 CentOS 的镜像：



正如你所看到的，基于 CentOS 的镜像列表很长，但是只有第一项是由官方提供的。

该镜像列表来自一个在 Docker 世界中被称作注册中心（Registry）的地方。Docker 注册中心就是一个对公共开放的简单镜像仓库。如果不想将自己的镜像推送到公共中心，那么可以选择运行自己的注册中心。

可以在以下链接找到更多相关信息：

https://docs.docker.com/registry/

在上图的表格中，有一列应该在第一时间就吸引了你的眼球，这一列就是 STARS（被标星的数量），该列表示了用户对指定镜像的评价。我们可以基于星数来减小对指定镜像的搜索范围，只需要使用-s 标记即可。

如果你运行下面的命令，将会看到标星数在 1000 以上的镜像列表：

```
docker search -s 1000 centos
```

请对你选择的镜像保持谨慎，因为并不能确保创建该镜像的用户不在其中暗藏恶意软件。

为了在本地机器上获取 CentOS 镜像，我们需要运行以下命令：

```
docker pull centos
```

该命令的输出如下所示：

```
➜ ~ docker pull centos
Using default tag: latest
latest: Pulling from library/centos

fa5be2806d4c: Pull complete
2bf4902415e3: Pull complete
86bcb57631bd: Pull complete
c8a648134623: Pull complete
Digest: sha256:8072bc7c66c3d5b633c3fddfc2bf12d5b4c2623f7004d9eed6aae70e0e99fbd7
Status: Downloaded newer image for centos:latest
```

一旦命令执行结束，再次运行 Docker 镜像，我们会发现 centos 已经出现在下面的列表中：

```
➜ ~ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
centos              latest              c8a648134623        3 weeks ago         196.6 MB
```

正如之前提到的，Docker 并不会使用整个操作系统，它只会使用一个剪裁版本，即只虚拟化操作系统的最后几层。你会发现镜像只有不到 200MB 大小，而 CentOS 的完整版是有好几个 GB 的。

运行容器

现在本机已经拥有了一个镜像的副本，是时候来运行它了：

```
docker run -i -t centos /bin/bash
```

该命令将会产生如下输出：

```
➜ ~ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
centos              latest              c8a648134623        3 weeks ago         196.6 MB
➜ ~ docker run -i -t centos /bin/bash
[root@debd09c7aa3b /]#
```

正如你所看到的，终端的提示信息变成了例如 `root@debd09c7aa3b` 这种形式，这意味着你已经身处容器之中了。

从现在开始，我们运行的每个命令都会在容器版本的 CentOS Linux 中执行。

下面是 Docker 的另一个有趣的命令：

```
docker ps
```

如果我们在宿主机的一个新的终端中运行该命令（不用退出运行中的容器），将会得到如下输出：

```
➜ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
62e7336a4627        centos              "/bin/bash"         8 minutes ago       Up 8 minutes                            prickly_bartik
➜ ~
```

该输出不言自明，这是一种可简单用于查看 Docker 容器的方式。

安装必要的软件

让我们在容器中安装 Node.js：

```
curl --silent --location https://rpm.nodesource.com/setup_4.x | bash -
```

该命令会拉取并执行 Node.js 的安装脚本。

该命令会产生如下输出：

```
[root@0ab9098011c0 /]# curl --silent --location https://rpm.nodesource.com/setup_4.x | bash -

## Installing the NodeSource Node.js 4.x LTS Argon repo...


## Inspecting system...

+ rpm -q --whatprovides redhat-release || rpm -q --whatprovides centos-release || rpm -q --whatprovides cloudlinux-release || rpm -q --whatprovides sl-release
+ uname -m

## Confirming "el7-x86_64" is supported...

+ curl -sLf -o /dev/null 'https://rpm.nodesource.com/pub_4.x/el/7/x86_64/nodesource-release-el7-1.noarch.rpm'

## Downloading release setup RPM...

+ mktemp
+ curl -sL -o '/tmp/tmp.bdco6IfiZY' 'https://rpm.nodesource.com/pub_4.x/el/7/x86_64/nodesource-release-el7-1.noarch.rpm'

## Installing release setup RPM...

+ rpm -i --nosignature --force '/tmp/tmp.bdco6IfiZY'

## Cleaning up...

+ rm -f '/tmp/tmp.bdco6IfiZY'

## Checking for existing installations...

+ rpm -qa 'node|npm' | grep -v nodesource

## Run `yum install -y nodejs` (as root) to install Node.js 4.x LTS Argon and npm.
## You may also need development tools to build native addons:
##   `yum install -y gcc-c++ make`

[root@0ab9098011c0 /]#
```

根据输出的指令，我们将完成 node 的安装：

```
yum install -y nodejs
```

强烈建议把开发工具也安装上，因为很多模块的安装流程都需要经历编译这一环节。让我们来安装开发工具：

```
yum install -y gcc-c++ make
```

一旦命令结束，我们便完成了在容器中运行 node 应用的准备工作。

保存修改

在 Docker 的世界里，镜像相当于针对特定容器的配置。我们可以将一个镜像作为模板，然后按需运行在各种容器中。然而，我们首先需要做的是保存之前对容器所做的修改。

如果你是一个软件开发者，你应该对 CVS、Subversion 及 Git 不陌生。Docker 也是基于它们的哲学构建的——一个容器可以被视为一个版本化的软件组件，针对该容器的修改也可以被提交。

为了实现这一点，我们先来运行以下命令：

```
docker ps -a
```

该命令会展示运行中的容器列表，如下图所示：

```
➜ ~ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
cf072Sa25148        centos              "/bin/bash"         11 seconds ago      Up 10 seconds                                   reverent_ritchie
62e7336a4627        centos              "/bin/bash"         15 minutes ago      Exited (0) About a minute ago                   prickly_bartik
7bbb50bb7236        centos              "/bin/bash"         16 minutes ago      Exited (0) 16 minutes ago                       jolly_raman
482d0ef324f2        centos              "/bin/bash"         22 minutes ago      Exited (127) 17 minutes ago                     adoring_hypatia
debd09c7aa3b        centos              "/bin/bash"         47 hours ago        Exited (137) 24 minutes ago                     nostalgic_stallman
ed948a19739b        centos              "/bin/bash"         47 hours ago        Exited (0) 47 hours ago                         jolly_mcclintock
c1f8550c09bb        1d073211c498        "/bin/bash"         10 weeks ago        Exited (0) 10 weeks ago                         fervent_torvalds
```

在我的机器上，运行着很多容器，其中比较有趣的是第二个，就是 Node.js 安装所在的容器。

现在，我们需要提交容器的状态来为之前的修改创建新的镜像。可以通过运行以下命令来完成它：

```
docker commit -a dgonzalez 62e7336a4627 centos-microservices:1.0
```

我们来对命令做一下说明：

- 标记 `-a` 表明了作者的身份，此处是 `dgonzalez`。
- 后面跟着的参数是容器 id。之前我们曾提到过，第二个容器对应的 ID 是 `62e7336a4627`。
- 第三个参数是新镜像的名字与标签的组合。如何区分镜像之间的微小变化会变得非常复杂，所以当我们需要处理相当多的镜像时，就会体会到该标签系统的强大。

该命令运行完需要几秒的时间。当命令结束后，命令的输出大致如下所示：

```
➜ ~ docker commit -a dgonzalez 62e7336a4627 centos-microservices:1.0
75d9f196b7b4181f41a09163d8177eefcc57649af1ccac9dbcc3af1e2a56bea6
```

以上信息表明，安装了我们指定软件的新镜像已经在镜像列表中了。再次运行 `docker images`，通过输出便可以确认这一点，如下图所示：

```
➜ ~ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
centos-microservices    1.0                 75d9f196b7b4        About a minute ago  306.4 MB
centos                  latest              c8a648134623        3 weeks ago         196.6 MB
```

为了基于新的镜像来运行容器，可以运行以下命令：

```
docker run -i -t centos-microservices:1.0 /bin/bash
```

该命令会带我们进入容器内的 shell，接着可以通过运行 `node -v` 来确认 Node. js 是否已安装，该命令会打印出 Node 的版本，在这个例子中会显示 4.2.4。

部署 Node.js 应用

现在，是时候尝试在容器内部署 Node.js 应用了。为了实现这一点，我们需要让本地机器的代码对 Docker 容器可见。

正确的方式是将本地目录挂载到 Docker 机器上。但是首先，我们需要创建一个小的应用以供容器稍后运行，如下所示：

```
var express = require('express');
var myApplication = express();

app.get('/hello', function (req, res) {
  res.send('Hello Earth!');
});

var port = 80;

app.listen(port, function () {
  console.log('Listeningistening on port '+ port);
});
```

这是一个简单的应用，它采用了 Express，主要功能是将字符串“`Hello Earth!`”呈现在浏览器中。如果在终端中运行它，并访问 `http://localhost:80/hello`，可以看到其结果。

现在，我们将在容器内运行这段代码。为了实现这一点，需要先将一个本地文件夹挂载为 Docker 容器的一个卷（volume），然后再运行它。

Docker 的灵感源自 DevOps，最近业界将系统管理员和开发这两类角色融合为了一个角色，统称为 DevOps。在 Docker 出现之前，每个公司都有其自有的一套用于部署应用和管理配置的方式，所以业界并没有公认能做好这些事情的最佳实践。

Docker 为所有公司提供了统一的部署方案。不管你的业务是什么，所有的事情都被简化成构建容器、部署应用，然后在合适的机器上运行容器。

假设现在应用所在的目录是/apps/test/。为了让它对容器可见，我们需要运行如下命令：

```
docker run -i -t -v /app/test:/test_app -p 8000:3000 centos-
microservices:1.0 /bin/bash
```

正如你所看到的，Docker 拥有非常详细的参数，让我们来做一下说明：

- 我们对标志-i 和-t 应该并不陌生。它们捕获输入并将其发送到一个分配的伪终端。
- 标志-v 我们第一次看到。它表明了本地机器目录挂载到容器卷的映射关系。在这个例子中，我们将本地机器的/apps/test 目录挂载到了容器的/test_app 目录。请注意其间的冒号，它将本地路径与容器路径进行了分隔。
- 标志-p 表明了本地机器端口与容器端口的映射关系。在这个例子中，我们通过 Docker 宿主机的 8000 端口来暴露容器的 3000 端口，所以访问 docker 宿主机的 8000 端口会最终访问到容器中的 3000 端口。
- centos-microservices:1.0 是镜像的名字和标记，该镜像是我们在前面的小节中创建的。
- /bin/bash 是我们希望在容器中执行的命令。/bin/bash 将会带我们进入系统提示。

如果一切进展顺利，我们将会进入容器的系统提示界面，如下图所示：

```
[root@c079d5f180da /]# cd /test_app/
[root@c079d5f180da test_app]# ls
node_modules  small-script.js
[root@c079d5f180da test_app]#
```

你可以在容器中看到，里面有一个叫作/test_app 的文件夹，它包含了之前编写的应用，我们将它命名为 small-script.js。

现在，是时候来访问应用了。但是，首先让我们来对 Docker 的工作方式做一下说明。

Docker 是采用 GO 编写的，GO 是由谷歌创建的一门现代语言，它提取了像 C++这样的编译型语言的好处，同时也兼具了像 Java 这样的现代语言的高级特性。

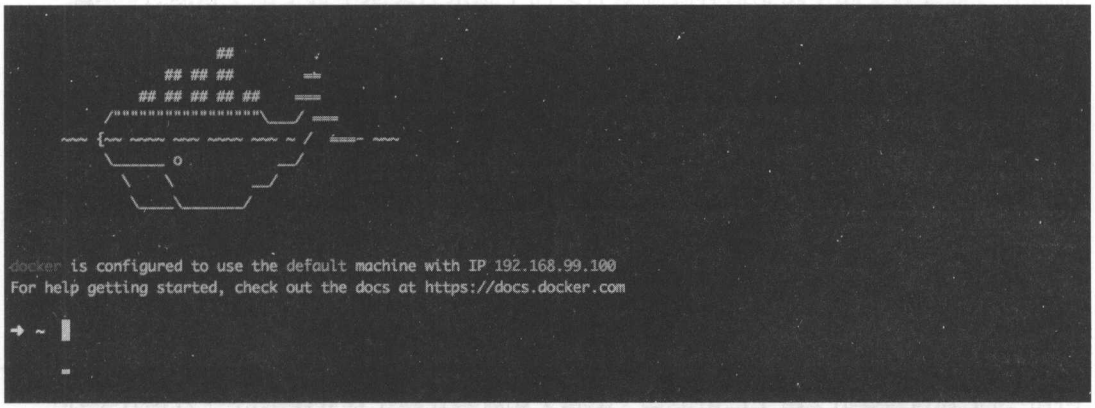
Go 很容易入门，却很难精通。Go 的哲学能为我们带来解释型语言的所有好处，相对于编译型语言来说，它减少了编译时间（整个语言可以在一分钟内编译完）。

Docker 使用了 Linux 内核的特定功能，所以 Windows 和 Mac 的用户必须要使用虚拟机才能运行 Docker 容器。该虚拟机之前叫作 boot2docker，而新的版本叫作 Docker Machine，它包含了更多的高级特性，比如在远程虚拟机部署容器。在这个实例中，我们将只会使用

本地的功能。

假定你已经运行了容器中位于/test_app/文件夹中的应用，且你所在的操作系统是Linux，那么只需访问 http://localhost:8000/，就可以看到该应用了。

当你使用的是 Mac 或 Windows 时，Docker 可以运行在 Docker Machine 或 boot2docker 中，而 IP 是虚拟机分配的，当 Docker 终端启动时，该 IP 会显示出来，如下图所示：



正如你所看到的，它的 IP 是 192.168.99.100，可以通过 http://192.168.99.100:9000/来访问该应用。

将 Docker 容器的创建过程自动化

如果你还记得，自动化是我们在前面章节中提过的最重要的概念之一。自动化是与微服务打交道时的关键。因为你不再是只操作一台机器，可能需要操作数十台机器，而这些工作迟早会塞满你的日常工作列表。

Docker 的设计者为了解决这类问题，允许用户通过编写脚本来创建容器。承载该脚本的文件被称为 Dockerfile。

如果你曾采用过 C 或 C++编写程序（即使只是在学校里接触过），那么应该对 Makefile 不会感到陌生。Makefile 文件是一段开发者用以指定自动化构建软件组件的脚本。来看看下面的例子：

```
all: my-awesome-app

my-awesome-app: main.o external-module.o app-core.o
  g++ main.o external-module.o app-core.o -o my-awesome-app
```

```
main.o: main.cc
  g++ -c main.cc


external-module.o: external-module.cc
  g++ -c external-module.cc


app-core.o: app-core.cc
  g++ -c hello.cc

clean:
  rm *.o my-awesome-app
```

上述的 Makefile 包含了一组任务及其执行时所需的依赖。举个例子，如果在包含 Makefile 的目录下执行 make clean，它将会删除可执行文件及所有以.o 结尾的文件。

Dockerfile 与 Makefile 是不同的，它并不是一组任务及其依赖（即使它们在概念上是相似的），它是一组用以从头开始构建容器并使之进入可用状态的指令。

让我们来看一个例子：

```
FROM centos
MAINTAINER David Gonzalez
RUN curl --silent --location https://rpm.nodesource.com/setup_4.x | bash -
RUN yum -y install nodejs
```

仅仅这几行就足以构建一个安装了 Node.js 的容器了。

让我们来对这些命令做一下说明：

- 首先，选择一个基础镜像。在本例中，我们使用了 centos。为此，我们需要使用 FROM 指令，并且后面紧跟镜像的名字。
- MAINTAINER 表明了容器作者的姓名。在本例中是 David Gonzalez。
- RUN 指令不言而喻，表示运行一个命令。在本例中，我们使用了两次 RUN：第一次是向 yum 添加仓库，第二次是安装 Node.js。

Dockerfile 可以包含一系列不同的指令，这些指令在官方文档中已讲述得非常清晰，但

是我们还是一起来看看最常用的几个指令（除了那些已经见过的）。

- CMD：该指令看上去与 RUN 非常相似，但是它实际上是在镜像构建过程结束后执行的。CMD 指令通常用于在容器初始化时启动应用。
- WORKDIR：该指令通常是结合 CMD 使用的，用于指定下一个 CMD 指令的工作目录。
- ADD：该指令用于将本地文件系统中的文件复制到容器实例中的文件系统。在前面的例子中，我们可以使用 ADD 来将应用从宿主机复制到容器中，并使用 CMD 来运行 npm install，然后再次使用 CMD 指令来运行应用。也可以使用 ADD 来从某个 URL 获取内容，然后复制到容器中的目标文件夹。
- ENV：该指令用于设置环境变量。举个例子，你只需向容器传递一个环境变量，便可以为上传文件指定一个专用的存储文件夹，如下所示：

```
ENV UPLOAD_PATH=/tmp/
```

- EXPOSE：该指令可用于向集群中的其他容器暴露指定端口。

你会发现，Dockerfile 的领域特定语言（DSL）已经相当丰富，可以使用它们尽情构建出各种需要的系统。互联网上充斥着数以百计的可参考实例，可帮助你构建出几乎任意的系统，比如：MySQL、MongoDB 和 Apache server 等。

强烈推荐通过 Dockerfile 来创建容器，因为它可以作为未来我们用来复制和变更容器的脚本所用，同样也可以让我们在无须人工介入的情况下完成自动化部署。

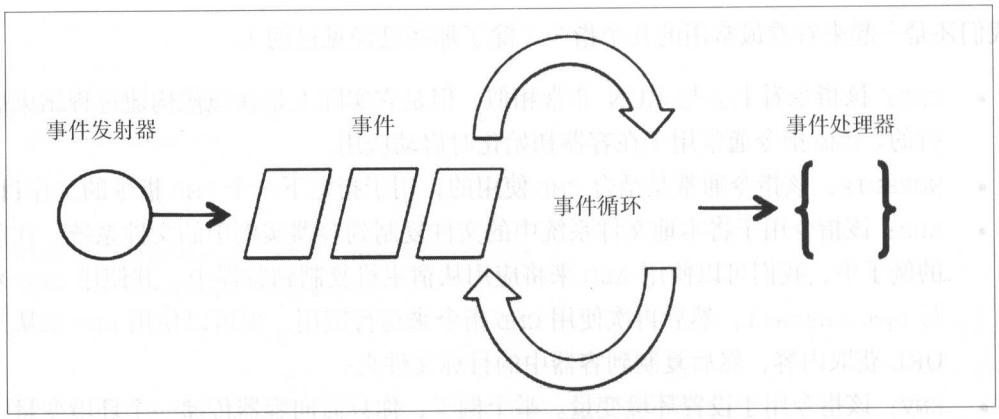
Node.js 事件循环——入门容易精通难

众所周知，Node.js 是一个采用单线程方式来运行应用的平台。然而，我们为何不采用多线程的方式来运行应用，从而利用上多核处理器的强大计算能力呢？

Node.js 是基于一个叫作 libuv 的库构建的。该库抽象了系统调用，为使用它的程序提供了异步编程接口。

我拥有深厚的 Java 背景，在 Java 中，所有事务都是同步的（除非你采用了非阻塞的库来进行编码），如果你向数据库发起一个请求，该线程会一直阻塞，直到数据库返回数据才会恢复。

这种方式通常也能正常工作，但是存在着一个需要令人关注的问题：阻塞线程所占用的资源其实可以先转而服务其他请求。Node.js 的事件循环如下图所示：



JavaScript 默认是一门事件驱动的语言。它会为执行的程序配置一系列事件处理器，这些事件处理器会对特定事件进行响应。而在完成配置后，它便开始等待事件行为的触发。

让我们来看一个熟悉的例子：

```
<div id="target">
  Click here
</div>
<div id="other">
  Trigger the handler
</div>
```

请看下面的 JavaScript 代码：

```
$( "#target" ).click(function() {
   alert( "Handler for .click() called." );
});
```

正如你所看到的，这是一个非常简单的例子。HTML 展示了一个按钮及一段使用了 jQuery 的 JavaScript 代码。当按钮被单击的时候，页面会展示一个警告弹窗。

关键的细节发生在按钮被单击的时候。

单击按钮是一个事件，经过 JavaScript 的事件循环，该事件会由 JavaScript 指定的特定处理器处理。

然而自始至终，仅有一个线程在执行这些事件。我们从未在 JavaScript 中谈论过并行（parallelism），而正确的说法应该是并发（concurrency）。因此，更为简洁地说，可以认为 Node.js 程序是高度并发的。

无论何时，你的应用将仅会由一个线程执行，在编码的时候请记住这一点。

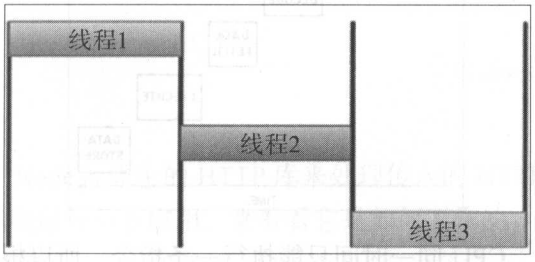
如果你曾使用过 Java 或 .NET，又或者使用过其他任意由线程阻塞技术设计并实现的语言或框架，或许就会注意到 Tomcat 是如何运行应用的，它会产生多个线程来监听请求。

在 Java 的世界里，每个线程都被称为 worker，它们负责全程处理一个来自特定用户的请求。在 Java 中，有一种数据结构可供我们善加利用。该数据结构叫作 ThreadLocal，它存储了当前线程的数据，这样一来可以便于我们在稍后恢复这些数据。这种存储类型之所以可以应付这类场景，是因为启动请求的线程最终会负责结束该请求，如果线程执行了任意的阻塞操作（例如读文件或访问数据库），它将会持续等待直到请求结束。

这通常并不是什么大问题，但是当你的软件重度依赖于 I/O 时，问题就会变得较为严重。

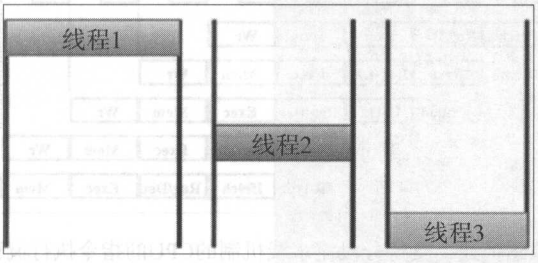
Node.js 的非阻塞模型的另一个优势是无须进行上下文切换。

当 CPU 从一个线程切换到另一个线程时，所有寄存器以及其他内存区域中的数据都会以栈的形式进行存储，这允许 CPU 可以切换到另一个新的线程的上下文，而该线程也拥有其自己的数据，如下图所示：



该图展示了理论上线程之间的上下文切换

该操作很花费时间，而这些时间却不能为应用所用，所以等同于直接被浪费掉了。在 Node.js 中，你的应用只会运行在一个线程上，所以在运行的时候并不会存在这类上下文切换的开销（其实上下文切换在背后也是存在的，只是对程序而言是不可见的）。如下图所示，我们可以看到现实世界中，CPU 切换线程时会发生什么：



该图展示了实际上线程之间的上下文切换情况（展示了那部分牺牲掉的空档时间）

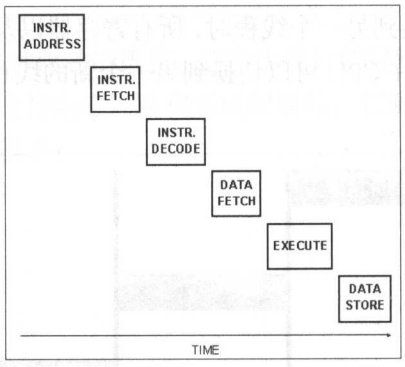
Node.js 应用的集群化

到现在为止，你已经知道了 Node.js 应用是如何工作的。当然，很多读者也许会有一个疑问，如果应用运行在单个线程上，那么在与现代多核处理器一起工作时会发生什么呢？

在回答这个问题之前，让我们先来看一下下面的场景。

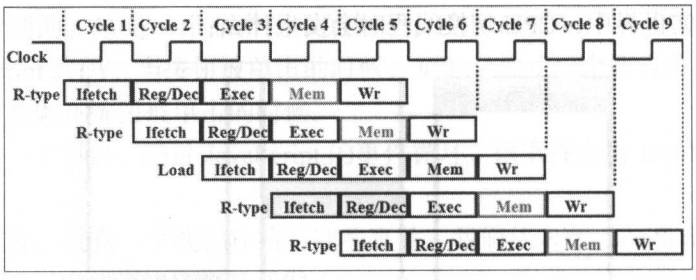
当我还是一个高中生的时候，CPU 领域就已经出现了一个很大的技术飞跃：分段机制（segmentation）。

这是首次在指令级别来尝试讲解并行。你或许已经意识到，CPU 会解释汇编指令，而每条指令是由一系列阶段组成的，如下图所示：



在 Intel 4x86 之前，CPU 同一时间只能执行一条指令，所以根据上图中的指令模型，任何 CPU 在每 6 个 CPU 周期内将只能执行一条指令。

然而，分段机制出现了，借助于一系列的中间寄存器，CPU 工程师们得以将指令的各个独立的阶段并行化。所以在最好的情况下，CPU 可以（或接近）在每个周期内都能执行一条指令，如下图所示：



该图描述了支持分段流水线机制的CPU的指令执行流程

该技术的进步催生了更快的 CPU，并为原生的硬件多线程打开了方便之门，从而进一

步催生了可以并发执行大量任务的现代多核处理器。但是当运行 Node.js 应用时，我们却只能使用一个核。

如果不对应用进行集群化，那么相比于其他平台，在利用多核 CPU 的优势方面就会有较严重的性能劣势。

然而，这一刻我们是幸运的。PM2 已经可以让我们对 Node.js 应用进行集群化，从而最大化地利用好 CPU。

同时，PM2 的一个重要优势是允许在不停机的情况下扩展应用的规模。

让我们以集群模式来运行一个简单的应用：

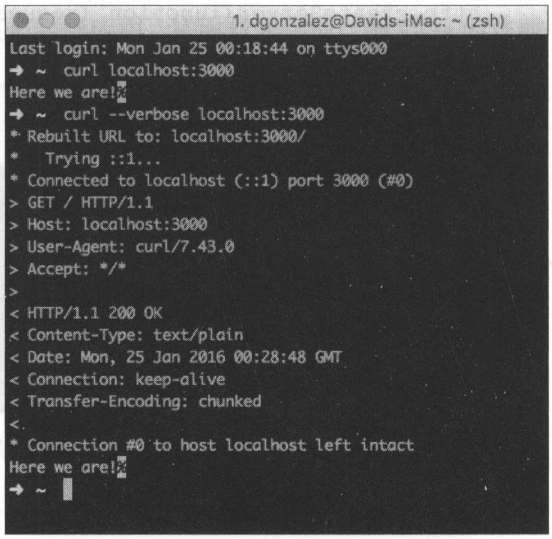
```
var http = require("http");
http.createServer(function (request, response) {
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });
  response.write('Here we are!')
  response.end();
}).listen(3000);
```

这一次我们使用了 Node.js 原生的 HTTP 库来处理传入的 HTTP 请求。

现在我们可以在终端运行一下应用，来看看它是如何工作的：

```
node app.js
```

虽然它什么也没有输出，但是我们可以通过采用 curl 来访问 URL http://localhost: 3000/，以便看见服务器的响应，如下图所示：



正如你所看到的，Node.js 接管了所有的 HTTP 处理，并且回复了代码中定义好的短语 `Here we are!`。

该服务相当简单，但是它所阐述的工作原理却与那些更加复杂的 Web 服务是一样的，所以我们可以通过将 Web 服务集群化从而避免遭遇瓶颈。

Node.js 拥有一个叫作 `cluster` 的库，通过它可以采用编程的方式来让应用集群化，如下所示：

```
var cluster = require('cluster');
var http = require('http');
var cpus = require('os').cpus().length;

// 此处会确认本进程是否是集群的master：根进程需要fork出所有执行Web服务的子进程
if (cluster.isMaster) {
  for (var i = 0; i < cpus; i++) {
    cluster.fork();
  }

  cluster.on('exit', function (worker, code, signal) {
    console.log("Worker " + worker.proces.pid + " has finished.");
  });
} else {
  // 如果是子进程，则会执行Web服务器
  http.createServer(function (request, response) {
    response.writeHead(200);
    response.end('Here we are!d\n');
  }).listen(80);
}
```

如果想要管理好集群化的应用实例，代码会变得非常复杂。就个人而言，我认为采用像 PM2 这样特定的软件可以简单地完成有效的集群化。

鉴于此，可以像下面这样通过 PM2 来运行应用：

```
pm2 start app.js -i 1
```

```
→ pm2-scale  pm2 start app.js -i 1
[PM2] Starting app.js in cluster_mode (1 instance)
[PM2] Done.
┌──────────┬────┬─────────┬───────┬────────┬─────────┬──────────┬──────────┐
│ App name │ id │ mode    │ pid   │ status │ restart │ uptime   │ memory   │ watching │
├──────────┼────┼─────────┼───────┼────────┼─────────┼──────────┼──────────┤
│ app      │ 0  │ cluster │ 24808 │ online │ 0       │ 0s       │ 17.637 MB│ disabled │
└──────────┴────┴─────────┴───────┴────────┴─────────┴──────────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app
```

在输出的命令中你可以看到-i 标记，在 PM2 中，该标记可用于指定应用计划使用的核数。

如果我们运行 pstree，可以看到系统中的进程树，请确认 PM2 是否只为应用运行了一个进程，如下图所示：

```
|-+= 23619 dgonzalez /Applications/Atom.app/Contents/Frameworks/Electron Framework.framework/Resources/crashpad_handler --database=/tmp/Atom Crashes --url=http://54.249.141.255:1127/post --handshake-fd=43
|-+= 23699 dgonzalez /Applications/Preview.app/Contents/MacOS/Preview -psn_0_13659398
|-+= 23764 dgonzalez /System/Library/PrivateFrameworks/HelpData.framework/Versions/A/Resources/helpd
|-+= 23767 dgonzalez /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -s mdworker -c MDSImporterWorker -m com.apple.mdworker.shared
|-+= 23818 dgonzalez /Applications/Skype.app/Contents/MacOS/Skype
|-+= 23930 dgonzalez /Applications/VLC.app/Contents/MacOS/VLC
|-+= 23952 root /usr/sbin/ocspd
|-+= 24185 dgonzalez /usr/local/Cellar/macvim/7.4-73_1/MacVim.app/Contents/MacOS/MacVim -MMNoWindow yes
|-+= 24147 dgonzalez /usr/local/Cellar/macvim/7.4-73_1/MacVim.app/Contents/MacOS/Vim -f -g app.js
|-+= 24191 dgonzalez /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -s mdworker -c MDSImporterWorker -m com.apple.mdworker.single
|-+= 24192 dgonzalez /System/Library/Frameworks/QuickLook.framework/Resources/quicklookd.app/Contents/MacOS/quicklookd
|-+= 24193 dgonzalez /System/Library/Frameworks/Quartz.framework/Frameworks/QuickLookUI.framework/Resources/QuickLookUIHelper.app/Contents/MacOS/QuickLookUIHelper
|-+= 24194 dgonzalez /System/Library/Frameworks/QuickLook.framework/Versions/A/Resources/quicklookd.app/Contents/XPCServices/QuickLookSatellite.xpc/Contents/MacOS/QuickLookSatellite
|-+= 24196 dgonzalez /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -s mdworker -c MDSImporterWorker -m com.apple.mdworker.single
|-+= 24200 dgonzalez /Applications/VirtualBox.app/Contents/MacOS/VirtualBox
|-+= 24202 dgonzalez /Applications/VirtualBox.app/Contents/MacOS/VBoxXPCOMIPCD
|-+= 24204 dgonzalez /Applications/VirtualBox.app/Contents/MacOS/VBoxSVC --auto-shutdown
|-+= 24516 dgonzalez PM2 v0.15.7: God Daemon
| \-+- 24808 dgonzalez node /Users/dgonzalez/Dropbox/Microservices with Node/Writing Bundle/Chapter 8/code/pm2-scale PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/Users/dgonzalez/Documents/software/activator-1.
\-+= 24817 dgonzalez /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -s mdworker -c MDSImporterWorker -m com.apple.mdworker.single
```

在这个例子中，我们只为应用运行了一个进程，所以只会为它分配一个 CPU 核。

虽然在本例中并未利用上 CPU 的多核能力，但是当应用中的算法抛出异常时，我们却能享受到应用自动重启的好处。

现在，我们将利用 CPU 中所有可用的核来运行应用，从而最大化地压榨 CPU 的性能。不过首先，需要先将集群停下来：

```
pm2 stop all
```

```
→ ~  pm2 stop all
[PM2] Stopping all
[PM2] stopProcessId process id 0
┌──────────┬────┬─────────┬─────┬─────────┬─────────┬──────────┬──────────┐
│ App name │ id │ mode    │ pid │ status  │ restart │ uptime   │ memory   │ watching │
├──────────┼────┼─────────┼─────┼─────────┼─────────┼──────────┼──────────┤
│ app      │ 0  │ cluster │ 0   │ stopped │ 0       │ 0        │ 0 B      │ disabled │
└──────────┴────┴─────────┴─────┴─────────┴─────────┴──────────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app
```

PM2停止所有服务后的状态

```
pm2 delete all
```

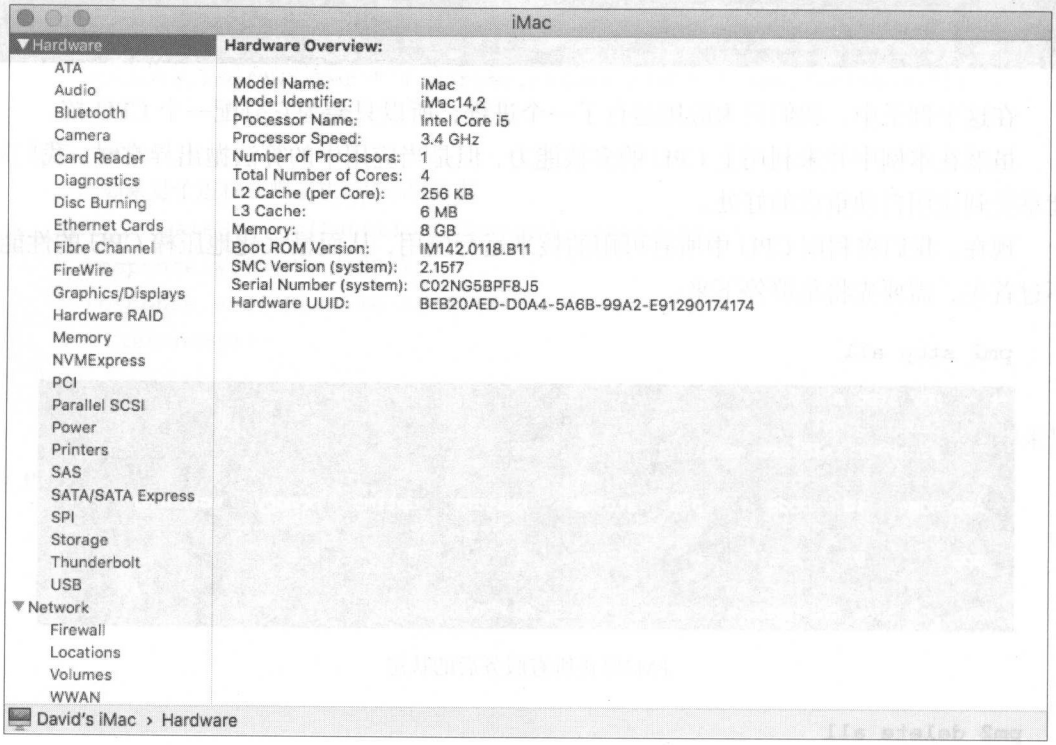
现在，可以利用所有的 CPU 核来重新运行应用了：

```
pm2 start app.js -i 0
```

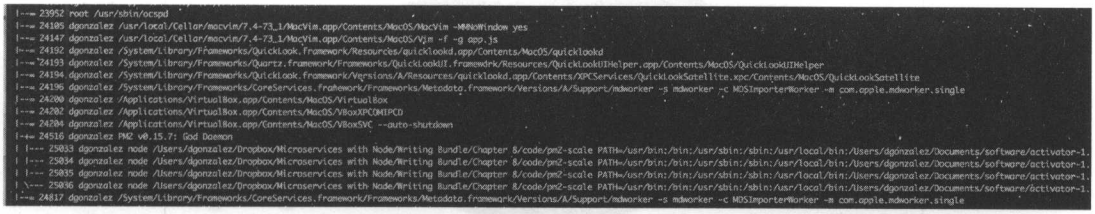
```
→ pm2-scale  pm2 start app.js -i 0
[PM2] Starting app.js in cluster_mode (0 instance)
[PM2] Done.
┌──────────┬────┬─────────┬───────┬────────┬─────────┬────────┬─────────┬──────────┐
│ App name │ id │ mode    │ pid   │ status │ restart │ uptime │ memory  │ watching │
├──────────┼────┼─────────┼───────┼────────┼─────────┼────────┼─────────┼──────────┤
│ app      │ 0  │ cluster │ 25033 │ online │ 0       │ 0s     │ 26.156 MB │ disabled │
│ app      │ 1  │ cluster │ 25034 │ online │ 0       │ 0s     │ 25.941 MB │ disabled │
│ app      │ 2  │ cluster │ 25035 │ online │ 0       │ 0s     │ 26.055 MB │ disabled │
│ app      │ 3  │ cluster │ 25036 │ online │ 0       │ 0s     │ 19.305 MB │ disabled │
└──────────┴────┴─────────┴───────┴────────┴─────────┴────────┴─────────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app
```

PM2展示了当前集群中运行的4个服务

PM2 能够猜测计算机中的 CPU 数，在这个例子中，我们使用的是一台 4 核的 iMac，如下图所示：



可以在 pstree 中看到，PM2 启动了 4 个进程，如下图所示：



当要对应用进行集群化时，对于应用如何规划所用 CPU 的核数有着一个不成文的规定，即该数字应该是总核数减去 1。

选择该数字背后的原因是因为操作系统也需要消耗一些 CPU，如果我们的应用占用了全部 CPU，一旦操作系统要处理一些其他任务，就会因为没有空闲的核而强制进行上下文切换，这会减慢应用的处理速度。

为应用增加负载均衡

有的时候，对应用进行集群化还不够，我们需要对应用进行水平扩展。

有了像亚马逊这样的云服务提供商，我们便有了很多对应用进行水平扩展的方法，而每个独立的提供商都实现了自有的一套拥有多种功能的解决方案。

我比较偏好的一种方式是采用 NGINX 来实现负载均衡。

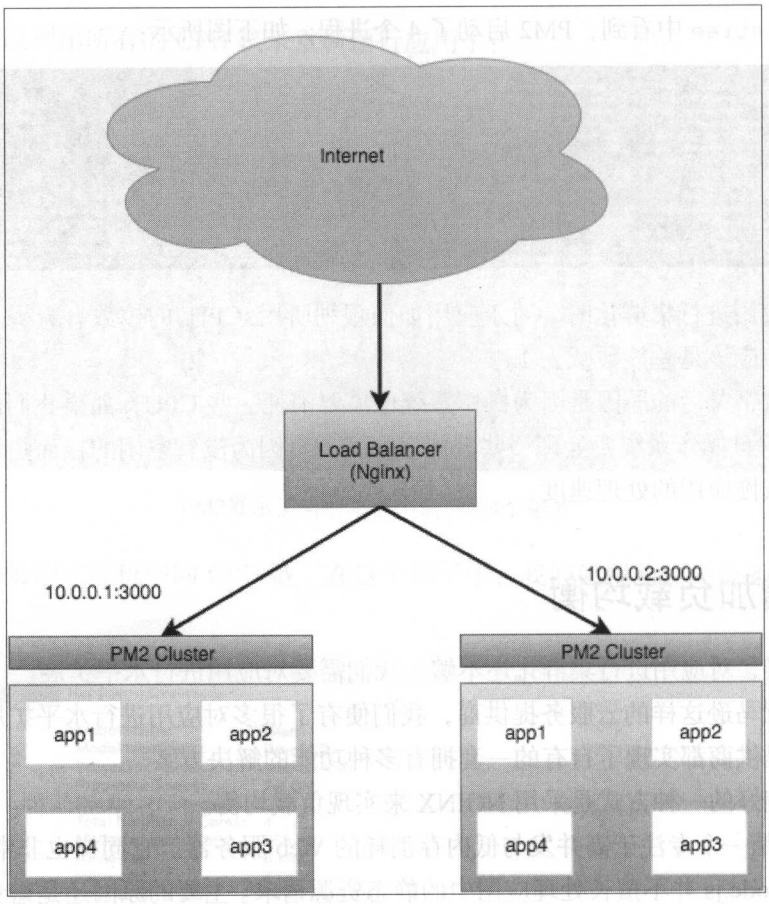
NGINX 是一个专注于高并发与低内存消耗的 Web 服务器。它同样也非常适合 Node.js 应用，因为 Node.js 并不擅长处理应用中的静态资源请求。主要的原因还是避免让应用处于压力之下，而让能更好处理某类任务的软件，比如 NGINX 来处理相应的任务（这是专职化的另一个例子）。

不管怎样，让我们来继续关注负载均衡。下图展示了 NGINX 作为负载均衡器的工作原理。

在图中，你可以看到，我们拥有两个 PM2 集群，并由一个 NGINX 实例来对它们进行负载均衡。

我们首先需要了解的是 NGINX 是如何管理配置的。

在 Linux 上，可以通过 yum、apt-get 或其他的包管理器来安装 NGINX。也可以从源码来编译构建 NGINX，除非你有特别的需求，否则还是推荐你使用包管理器。



默认情况下，主配置文件位于/etc/nginx/nginx.conf，内容如下所示：

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
  worker_connections  1024;
}

http {
```

```
    include                   /etc/nginx/mime.types;
    default_type               application/octet-stream;

    log_format         main '$remote_addr - $remote_user [$time_local]
      "$request" '
      '$status $body_bytes_sent "$http_referer" '
      '"$http_user_agent" "$http_x_forwarded_for" '
      '$request_time';

    access_log                /var/log/nginx/access.log  main;
    server_tokens             off;
    sendfile                  on;
    #tcp_nopush               on;
    keepalive_timeout         65s;
    send_timeout              15s;
    client_header_timeout     15s;
    client_body_timeout       15s;
    client_max_body_size      5m;
    ignore_invalid_headers    on;
    fastcgi_buffers           16 4k;
    #gzip                     on;
    include                   /etc/nginx/sites-enabled/*.conf;
}
```

该文件相当简单明了，它指定了 worker（用于服务请求的进程）的数量、错误日志的路径、单个 worker 可拥有的活跃连接数以及最后的 HTTP 配置。

最后一行最值得关注：我们告知 NGINX 必须将`/etc/nginx/sites-enabled/*.conf`作为潜在的配置文件。

有了这段配置之后，任何在该指定目录下的以`.conf`结尾的文件都将成为 NGINX 配置的一部分。

你可以看到，默认的配置文件已经有了，我们来修改一下该文件：

```
http {
  upstream app {
    server 10.0.0.1:3000;
    server 10.0.0.2:3000;
  }
```

```
    server {
      listen 80;
      location / {
        proxy_pass http://app;
      }
    }
  }
```

这便是我们需要用于构建负载均衡器的所有配置。下面来逐项进行说明：

- `upstream app` 指令用于创建一组叫作 `app` 的服务。在这个指令内，我们指定了两个在之前插图中所展示的服务。
- 在 `server` 指令中指定了 NGINX 会在 `80` 端口监听所有请求，并将请求传递给叫作 `app` 的 upstream 服务组。

那么，NGINX 是如何决定将请求发送到哪台机器的呢？

在这个例子中，我们可以为如何传递负载指定策略。

默认情况下，如果没有配置特定的负载均衡方法的话，NGINX 会使用 round robin。

如果使用 round robin 的话，需要牢记一件事，即应用应该是无状态的，因为请求无法每次都命中同一台机器，所以如果在某台机器上保存了状态，那么在后续的调用中会由于命中的可能是另一台机器而无法获取之前的状态。

round robin 是用于将队列中的作业分发给一组 worker 的最基本的方式，通过不断地轮转分发，每个节点都会获得近似数量的请求。

还可以使用其他机制来传播负载，如下所示：

```
upstream app {
  least_conn;
  server 10.0.0.1:3000;
  server 10.0.0.2:3000;
}
```

least connected（最少连接），正如名字所表明的，会将请求发送给当前连接数最少的节点，这样可以在所有节点之间均等地分发请求：

```
upstream app {
  ip_hash;
  server 10.0.0.1:3000;
```

```
    server 10.0.0.2:3000;
}
```

IP hashing 是最有趣的一种用于分发负载的方式。如果你曾跟任意的 Web 应用打过交道，对会话（session）的概念应该不会陌生，它几乎在任何应用中都有出现。为了记住用户是谁，浏览器会向服务器发送 cookie，利用 cookie 可以在内存中存储用户的身份信息以及他或她需要或可以访问的数据。如果采用其他负载均衡类型，那么问题就来了，我们无法保证每次都命中同一台机器。

举个例子，如果使用“最少连接”作为均衡策略，我们的第一次负载会命中某一个服务器，但是在后续的重定向中可能会命中不同的服务器，这会导致由于第二台服务器并不知道用户是谁而无法正确地展示用户信息。

采用 IP hashing 后，负载均衡器会为指定 IP 计算出一个 hash 值。该 hash 值会以某种方式生成一个 1 到 N 之间的数字，其中 N 是服务器的数量，这样一来，只要 IP 保持不变，用户就会总被重定向到同一台机器。

我们还可以为负载均衡器开启权重的功能，如下所示：

```
upstream app {
  server 10.0.0.1:3000 weight=5;
  server 10.0.0.2:3000;
}
```

这将会导致每 6 个请求里有 5 个请求被定向到第一台服务器，而有 1 个请求会被定向到第二台机器。

一旦选择好负载均衡的方式，便可以通过重启 NGINX 来使变更生效。但是首先，我们会像下图所示那样对这些配置进行校验：

```
vagrant@dgonzalez-vagrant ~ $ sudo /etc/init.d/nginx configtest
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
vagrant@dgonzalez-vagrant ~ $
```

你可以看到，对配置进行测试确实可以帮助我们避免由于配置错误而导致的故障。

一旦 NGINX 通过 `configtest`，便可以保证 NGINX 能够在配置没有语法错误的情况下完成 `restart/start/reload` 操作，如下所示：

```
sudo /etc/init.d/nginx reload
```

重新加载将会平滑等待老线程完成工作，然后重新加载配置并将新的请求路由到新配

置生效后的工作线程。

如果你对学习 NGINX 很感兴趣的话，下面的官方文档会对你很有帮助：

```
http://nginx.org/en/docs/
```

NGINX 的健康检查

健康检查是负载均衡器最重要的一项活动：如果某个节点出现了严重的硬件故障会发生什么？是不是就不能处理更多的请求了？

在这个例子中，NGINX 自带了两种类型的健康检查：被动检查和主动检查。

被动健康检查

在这里，NGINX 是作为反向代理来进行配置的（正如我们在上一节所做的）。它会将一个来自上游服务器的特定类型的响应作为反馈。

如果上游服务器的响应发生错误，NGINX 会将该节点标记为“错误”，并会将它从负载均衡器中摘除一段时间后再引入。由于具备了这样的策略，NGINX 将会不断地将出错的节点从负载均衡器中摘除，所以故障的次数也会大大减少。

还有一些其他的配置参数，例如 `max_fails` 或 `fail_timeout`，可分别用于配置当某个节点出现了多少次故障或请求时间超时多久时将其标记为无效节点。

主动健康检查

主动健康检查与被动健康检查不同，它会主动向上游服务器发起连接以检查它们是否能为测试请求做出正确的响应。

下面是一段最简单的用来在 NGINX 中进行主动监控的配置：

```
http {
  upstream app {
    zone app test;
    server 10.0.0.1:3000;
    server 10.0.0.2:3000;
  }
  server {
    listen 80;
    location / {
```

```
        proxy_pass http://app;
        health_check;
      }
    }
}
```

在配置文件中多了两行新的内容，如下所示。

- `health_check`：该项用以启用主动健康检查。默认配置会每 5 秒向 `upstream` 配置项指定的主机和端口发起连接。
- `zone app test`：该项是当启用健康检查时，NGINX 中的必配项。

我们拥有非常多的选择来配置更加具体的健康检查，它们都在 NGINX 中可配，通过结合使用这些健康检查的选项可以满足不同用户的需求。

小结

在本章中，我们领略了各种可用来部署微服务的技术。到现在为止，你已经知道了如何以一种特定的方式来构建、部署及配置软件组件，通过这种方式我们得以兼容并包多样化的各种技术。本书的目标是为你提供必备的思想从而帮助你开始使用微服务，并且让所有读者都能知道如何找到自己所需的信息。

就个人而言，我曾努力尝试找到一本能提供有关微服务整个生命周期各个方面相关信息的图书，而如今我真切地希望本书能填补这方面的空白。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

最近几年，微服务作为一剂应对单块架构的良方开始崭露头角，并逐渐成为构建企业级应用的首选方式。通过本书，我们将挖掘并利用Node.js的强大功能来快速开发出可在小型Docker容器中运行的轻量级微服务，并向你展现以这种方式部署及扩展服务所带来的愉悦感。

我们将会介绍Seneca和PM2等可用于实现Node.js微服务的各种工具，帮助理解并避免在实践微服务过程中可能遇到的常见陷阱。大家将学会如何通过最佳的日志和监控工具来掌控自己的应用。

在完成本书的学习后，你将理解如何构建具备良好测试覆盖率的高质量的微服务，以及在构建微服务时所需的有关Node.js的所有概念。

本书读者对象

本书适合具备基本服务端开发知识但不了解如何在Node.js应用中实现微服务的Node.js开发者。同时，它对使用Java和C#等其他语言的开发者也有所裨益。

你将会从本书中学到：

- 理解Node.js模块并掌握在与微服务打交道时的最佳实践
- 将现有的单块系统重新架构成面向微服务的软件
- 使用Seneca和Node.js构建出健壮且可伸缩的微服务
- 对微服务进行隔离测试，从而创建出可靠的系统
- 使用PM2部署并管理微服务
- 监控微服务的健康状况（CPU、内存以及I/O）

[PACKT] open source* PUBLISHING community experience distilled

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：张春雨
责任编辑：刘　舫
封面设计：吴海燕

上架建议：容器/运维

ISBN 978-7-121-30524-5
9 787121 305245 >

定价：69.00元